Universidade de Aveiro
2013
Departamento de Eletrónica,
Telecomunicações e Informática

Hugo Miguel Silva Pereira

Ambiente de execução distribuída de aplicações


Universidade de Aveiro
2013
Departamento de Eletrónica,
Telecomunicações e Informática

Hugo Miguel Silva Pereira

Ambiente de execução distribuída de aplicações

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor António Rui de Oliveira e Silva Borges, Professor Associado do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri

presidente

Prof. Dr. Tomás António Mendes Oliveira e Silva
Professor Associado da Universidade de Aveiro

Prof. Dr. António Rui de Oliveira e Silva Borges
Professor Associado da Universidade de Aveiro

Prof. Dr. Pedro Lopes da Silva Mariano
Professor Auxiliar Convidado do Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa

agradecimentos

Ao Professor António Borges, orientador da tese, pela disponibilidade, apoio e confiança perante os obstáculos.

À minha família, namorada e colegas de curso pelo apoio ao longo do curso.

palavras-chave

máquina virtual paralela, *cloud computing*, nó de processamento, execução distribuída de aplicações.

resumo

Com o aumento da banda de transmissão e aumento dos recursos ligados à rede, na maioria do tempo subaproveitados, torna-se interessante disponibilizar um conjunto de serviços que rentabilizem a sua utilização.

O objetivo deste trabalho é especificar tal sistema e efetuar um estudo de viabilidade, abordando questões como a disponibilidade, robustez e heterogeneidade dos recursos. Este estudo implica a implementação do sistema, e a demonstração do seu correto funcionamento, usando como recursos, os sistemas computacionais disponíveis no Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

O sistema desenvolvido neste projeto visa disponibilizar um ambiente de execução distribuída de aplicações. Como tal, os recursos que se pretende rentabilizar são o poder de processamento e a capacidade de armazenamento dos sistemas computacionais. O sistema é responsável por agregar os recursos e disponibilizá-los como um único sistema computacional paralelo.

O sistema foi implementado e validado, permitindo a execução de aplicações. Tendo em conta os testes efetuados, pode-se concluir que o sistema é estável e robusto, sendo capaz de resistir a falhas de múltiplos recursos em simultâneo.

keywords

Parallel virtual machine, cloud computing, processing node, distributed execution of applications.

abstract

With the increase of the transmission bandwidth and the increase of the resources connected to the network, underexploited most of the time, it is interesting to provide a set of services to increase their use.

The objective of this work is specify such a system and make a viability study, addressing issues such as availability, resilience and resource heterogeneity. This study involves the implementation of the system, and proof of its correct operation, using as resources, computer systems available in *Departamento de Eletrónica, Telecomunicações e Informática* of *Universidade de Aveiro*.

The system developed in this project aims to provide an execution environment for distributed applications, as such, the resources intended to make the most of are the processing power and the storage capacity of computer systems. The system is responsible to aggregate resources and to make them available as a single parallel computing system.

The system was implemented and validated, allowing the execution of multiple applications in parallel. Taking into account the tests that were carried out, it can be shown that the system is stable and robust, being able to withstand the simultaneous failure of multiple computing nodes.

Índice

1. Introdução

1.1. Motivação

Durante a última década ocorreram dois acontecimentos que impulsionaram a área de Sistemas Distribuídos: o aumento de recursos ligados à rede e o aumento das taxas de transmissão. Falar em recursos ligados à rede é o mesmo que referir sistemas computacionais, quer sejam eles computadores pessoais ou dispositivos móveis, com ligação à internet.

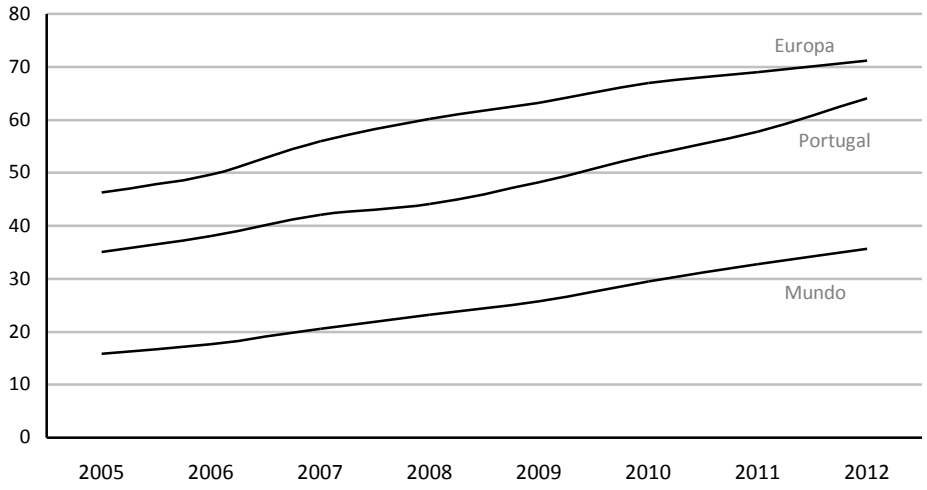

Figura 1.1 Percentagem de indivíduos que usam a internet [ITU, 2013]

Comparando o ano de 2005 com o ano de 2012, pode verificar-se que a percentagem de indivíduos que utilizam a internet em todo o mundo aumentou quase para o dobro, tendo-se registado em 2005 18,4% e, em 2012 35,7%. Na Europa a percentagem é bastante maior, sendo em 2012 71,2%. Portugal encontra-se abaixo da média europeia com 64% [ITU, 2013]. O aumento observado deveu-se essencialmente à generalização do uso de computadores pessoais e à divulgação de *smartphones* e *tablets*. Por esse motivo, os fornecedores de internet sentiram a necessidade de aumentar as taxas de transmissão para conseguirem suportar o consequente aumento do tráfego. O aparecimento de novas tecnologias de acesso à internet, como é o caso da fibra ótica, permitiu assegurar esse aumento.

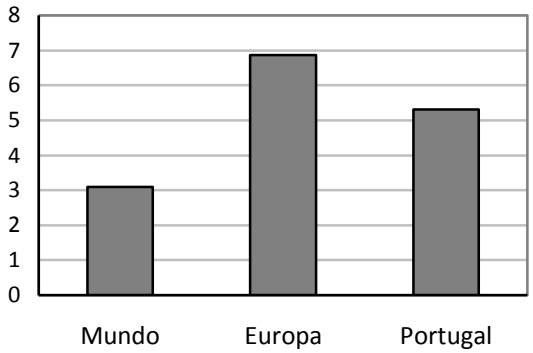

Figura 1.2 Velocidade média da ligação à Internet (Mbps) no 1º trimestre de 2013 [Akamai, 2013]

No primeiro trimestre de 2013 a velocidade média das taxas de transmissão a nível mundial rondava os 3,1 Mbps. No mesmo período, na Europa a média era de 6,86 Mbps e, em Portugal um pouco abaixo com 5,3 Mbps [Akamai, 2013].

Originou-se assim na atualidade, um subaproveitamento de recursos. Uma vez que a população em geral não tem interesse em rentabilizar os recursos dos seus computadores pessoais e dispositivos móveis, os potenciais interessados na rentabilização de recursos serão instituições e empresas que os possuam em elevada quantidade. Um exemplo de instituição com este tipo de problema é a Universidade de Aveiro.

A universidade possui cerca de 10500 computadores dispersos por todo o campus universitário [UA, 2011]. Esses computadores são utilizados para diversos fins, nomeadamente lecionação de aulas, investigação, administração e consulta livre. Como tal, existem vários computadores que estão em constante utilização e possuem os seus recursos já aproveitados. No entanto, existem outros que são utilizados em períodos bem definidos, como é o caso dos computadores utilizados para a lecionação de aulas e consulta livre, e que se encontram disponíveis noutros momentos.

O objetivo desta dissertação é propor uma rentabilização desses recursos, e do estabelecimento de um conjunto de serviços que permitam que trabalhadores/alunos da universidade com necessidade de recursos computacionalmente intensivos, lhes possam aceder e usufruir da forma mais eficiente possível.

1.2. Enquadramento

1.2.1. Mar de recursos

Os recursos considerados relevantes neste documento são o poder de processamento e capacidade de armazenamento (memória principal e memória de massa) das plataformas *hardware*.

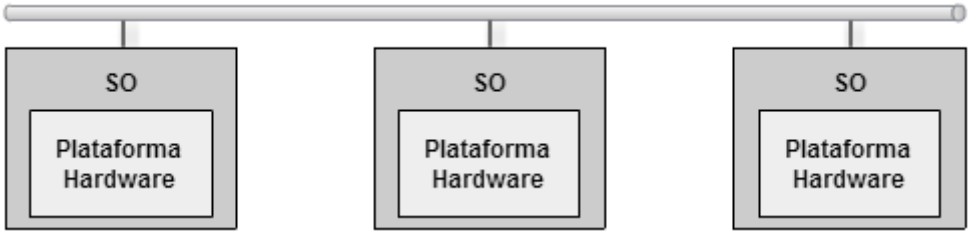

Figura 1.3 Abstração da plataforma *hardware* fornecida pelo sistema operativo

Tal como a figura anterior ilustra, os recursos da plataforma de *hardware* são normalmente encapsulados por um sistema operativo (SO), fornecendo uma abstração que permite a sua manipulação de uma forma mais simples. Este sistema operativo pode variar de máquina para máquina, sendo os mais utilizados atualmente em computadores e servidores o Microsoft Windows, o Mac OS e o Linux. Ou seja, a máquina virtual fornecida não é uniforme, o que exige que seja complementada por uma camada de transparência adicional.

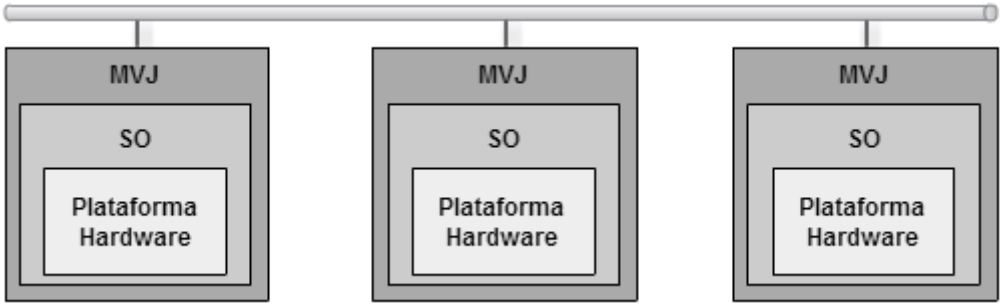

Figura 1.4 Abstração da plataforma *hardware* fornecida pela máquina virtual de Java

Para tal, recorreu-se à máquina virtual de Java [JVM, 2013], um ambiente de execução que pode ser instalado sobre vários sistemas operativos. Assim, a MVJ torna o acesso aos recursos das máquinas uniforme e independente das características específicas da plataforma *hardware* e do sistema operativo utilizados.

Apesar do acesso aos recursos ser homogéneo a localização deles encontra-se dispersa pelas múltiplas plataformas *hardware,* tonando necessário adicionar uma camada *middleware* para que o agregado de sistemas computacionais interligado seja percebido como uma entidade computacional única. Permitindo assim que o acesso aos recursos seja transparente.

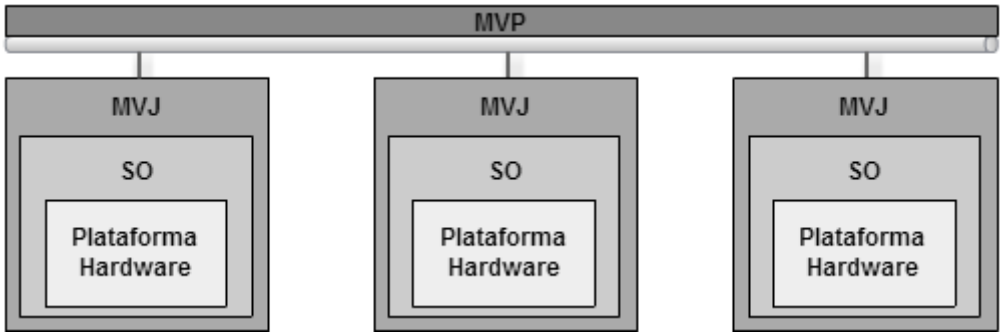

Figura 1.5 Abstração do agregado de plataformas hardware, percebido como uma entidade computacional única através da máquina virtual paralela

Essa camada é designada por máquina virtual paralela [PVM, 1994], um sistema *software* que agrega um conjunto de recursos e os disponibiliza como um único sistema computacional paralelo. Assim, para além da localização dos recursos ser encapsulada, estes são usufruídos concorrentemente pelas diversas aplicações. Para isso, é necessário respeitar o modelo de paralelização descrito a seguir.

1.2.2. Modelo de paralelização

O modelo de paralelização especifica a composição que as aplicações devem respeitar para que a sua execução seja o mais eficiente possível. Uma aplicação é a execução de uma tarefa que é composta por subtarefas independentes, que serão processadas por recursos independentes. Estas subtarefas consomem dados de entrada e produzem dados de saída (resultados), e podem ser organizadas de duas formas distintas: em série e em paralelo.

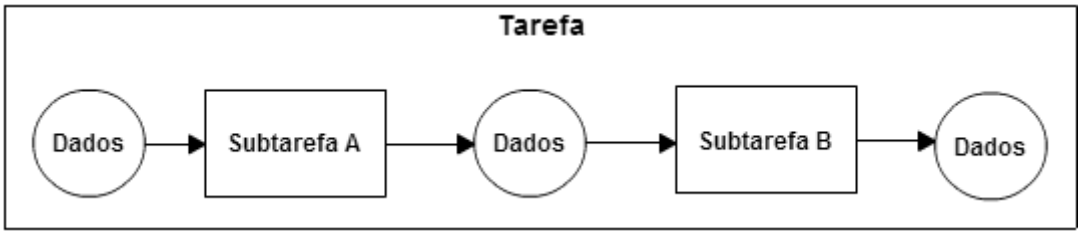

Figura 1.6 Tarefa com duas subtarefas em série

Para duas tarefas se encontrarem em série, uma subtarefa deve consumir resultados produzidos pela outra subtarefa. Seguindo o exemplo da figura anterior, a subtarefa A está em série com a subtarefa B, o que significa que a subtarefa B só pode ser processada depois da subtarefa A.

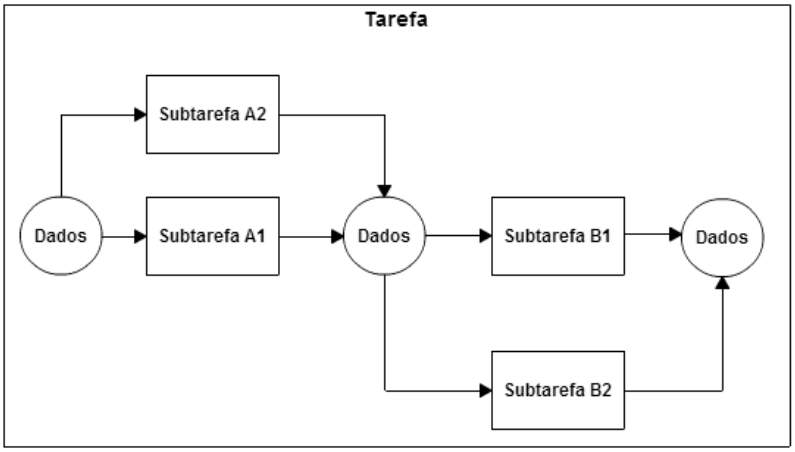

Figura 1.7 Tarefa com quatro subtarefas em série e paralelo

Para duas ou mais subtarefas se encontrarem em paralelo, estas devem ler da mesma fonte de dados. No exemplo representado na figura anterior, as subtarefas A1 e A2 são paralelas entre si, bem como as subtarefas B1 e B2. Quando duas ou mais subtarefas são paralelas significa que todas podem ser executadas em simultâneo, o que torna mais rápida a execução de uma tarefa. Através deste modelo de paralelização, seguindo ainda o exemplo da figura anterior, dois nós de processamento podem processar simultaneamente as subtarefas A1 e A2 usufruindo assim dos recursos de dois nós e, demorando menos tempo do que se apenas um nó processasse ambas as subtarefas.

1.3. Estrutura da dissertação

A presente dissertação pretende descrever o produto desenvolvido sobre o tema em análise, sendo constituída por quatro capítulos.

No capítulo 1 é feita uma pequena introdução ao tema em análise referindo os motivos que levaram a desenvolver este produto, os objetivos do mesmo e ainda, uma breve explicação de alguns conceitos associados.

O capítulo 2 refere-se ao estado de arte e pretende ser uma compilação da informação existente sobre os principais temas que envolvem o produto em causa. Sendo eles a evolução dos sistemas distribuídos (modelos e arquiteturas), dando mais relevância ao modelo utilizado no produto desenvolvido: "*Cloud Computing*".

O capítulo 3 descreve em detalhe o produto desenvolvido, incluindo a arquitetura de um nó de processamento, a organização da MVP, API de acesso dos serviços web, e ainda o cliente desenvolvido. São também justificadas todas as decisões tomadas ao longo do desenvolvimento do produto.

O capítulo 4 descreve alguns testes realizados sobre o sistema e fornece os resultados obtidos.

No capítulo 5 são apresentadas as conclusões, indicando outras opções que poderiam ter sido utilizadas no desenvolvimento do produto, bem como sugeridos futuros projetos que continuem o desenvolvimento do mesmo, e situações de aplicação.

2. Estado de arte – Sistemas Distribuídos

"Um sistema distribuído é a agregação de múltiplos computadores independentes que aparentam ser para os seus utilizadores um único computador" [Tanenbaum & Steen, 2006]. Ao longo do tempo foram surgindo vários modelos de sistemas distribuídos consoante as necessidades das empresas e das instituições.

2.1. Cluster Computing

Cluster Computing foi dos primeiros modelos de sistemas distribuídos a surgir, tornando-se bastante popular por ser simples e barato de implementar. É apenas necessário ligar numa rede local de alta velocidade um conjunto de computadores simples e semelhantes, quer em termos de *hardware*, quer em termos de *software*.

2.1.1. Caracterização

Existem algumas implementações de *cluster*, sendo uma das mais conceituadas e conhecidas o *Beowulf* [Sterling, 2001].

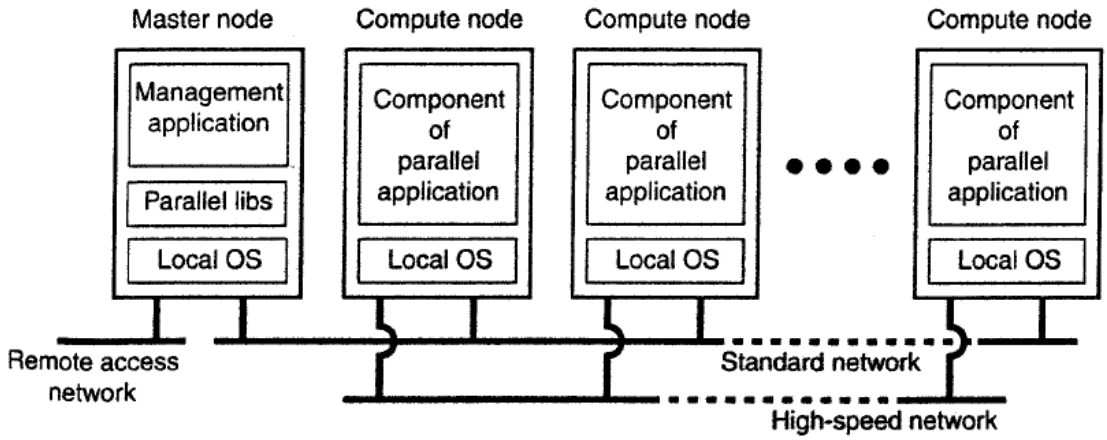

Figura 2.1 Arquitetura de *Beowulf* [Tanenbaum & Steen, 2006]

Nesta implementação um dos computadores do agregado é designado por *Master* e as suas funções são controlar os restantes computadores e providenciar a interface do sistema. Desta forma o *Master* possui a camada de *middleware* responsável pela execução de aplicações (desde a reserva de recursos até à manutenção da fila de espera) e pela gestão do agregado (deteção e recuperação de falhas). Já os restantes nós apenas necessitam de um sistema operativo [Tanenbaum & Steen, 2006].

2.1.2. Aspetos críticos

As duas principais vantagens deste modelo são a tranparência e a escalabilidade. Como o *Master* providencia a interface do sistema e controla os restantes computadores, os utilizadores não possuem conhecimento do número de computadores que compõem o agregado, nem as suas localizações ou estado. Para aumentar ou reduzir a capacidade do sistema, basta adicionar ou retirar computadores do agregado, respetivamente. No entanto, a escalabilidade do sistema é limitada pelas dimensões das instalações em que os computadores se encontram, uma vez que estes necessitam de estar ligados na mesma rede local.

Apesar das vantagens do modelo *Cluster Computing*, existem algumas limitações. Requisitos como a homogeneidade do agregado e a ligação dos computadores através de uma rede local restringem as opções das instituições e empresas na aquisição de equipamentos e no local da sua instalação. No entanto, mais grave que estas limitações é a vulnerabilidade do sistema, pois se o *Master* falhar todo o sistema falha (*Single Point of Failure*).

2.2. Grid Computing

Grid Computing surge da necessidade de encontrar soluções para as limitações e vulnerabilidades que o *Cluster Computing* apresentava.

2.2.1. Caracterização

Para ajudar na distinção do que é *Grid Computing* foram definidos três requisitos que devem ser respeitados [Foster, 2002]:

1) deve coordenar recursos que não se encontram sobre o mesmo domínio, isto é, não existe um controlo centralizado;

2) deve usar protocolos e interfaces *standard* e abertas (por exemplo, protocolos de autenticação, acesso e descoberta de recursos);

3) deve disponibilizar qualidades de serviço não triviais (por exemplo, disponibilidade de recursos, segurança, desempenho, etc.).

Os serviços e protocolos de *Grid Computing* são distribuídos por 5 camadas: *Fabric*, *Connectivity*, *Resource*, *Collective* e *Application*.

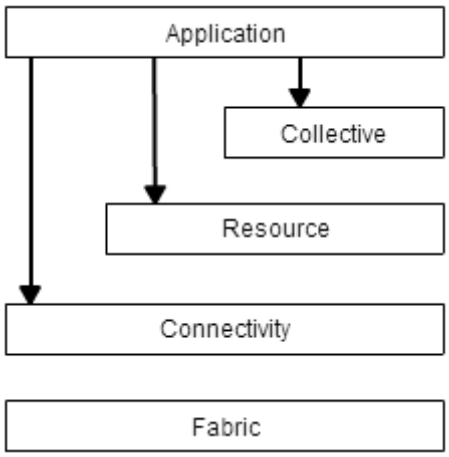

Figura 2.2 Arquitetura protocolar de *Grid Computing* [Foster, Zhao, Raicu, & Lu].

A camada *Fabric* disponibiliza acesso a diferentes tipos de recursos como armazenamento e processamento. A camada *Connectivity* define os protocolos de comunicação e autenticação para um acesso seguro aos recursos. A camada *Resource* estabelece os protocolos para publicação e descoberta de recursos, monitorização, pagamento, entre outros. A camada *Collective* monitoriza a interação entre recursos. Por último, a camada *Application* é um ambiente virtual composto pelas aplicações dos utilizadores em execução [Foster, Zhao, Raicu, & Lu] [Tanenbaum & Steen].

2.2.2. Aspetos críticos

Neste modelo nada é assumido sobre os computadores envolvidos, quer em termos de *software*, quer em termos de *hardware*. De facto, os computadores podem estar em redes diferentes, serem geridos por instituições diferentes e podem respeitar políticas diferentes. Assim os requisitos como a homogeneidade do sistema e a ligação dos computadores através de uma rede local são eliminados, retirando as restrições impostas pelo *Cluster Computing* sobre as instituições e empresas. Para além de retirar as restrições, possibilita a partilha de recursos e soluções de problemas entre diferentes instituições [Foster, Zhao, Raicu, & Lu]. Outra vantagem da *Grid* é a vulnerabilidade *Single Point of Failure* ter sido eliminada, uma vez que não existe um controlo centralizado sobre os recursos.

2.3. *Cloud Computing*

Com a generalização do uso dos computadores pessoais, algumas instituições que disponibilizavam serviços para o público em geral sentiram a necessidade de aumentar os seus recursos para satisfazerem as necessidades dos seus clientes. No entanto, a procura por esses serviços não é constante, levando a um subaproveitamento dos recursos nos períodos em que a

procura é inferior. Por esse motivo surgiu um novo paradigma na área de sistemas distribuídos, *Cloud Computing*.

2.3.1. Caracterização

O objetivo é manter os recursos o mais ocupados possível. Apesar de o termo *Cloud Computing* ter aparecido há alguns anos a sua definição ainda continua a alterar e a evoluir, não se tendo chegado a um consenso na comunidade. Em 2011 o "*National Institute of Standards and Technology*" (NIST) definiu *Cloud Computing* como um modelo que permite aceder a um conjunto de recursos partilhados, como aplicações, serviços e dispositivos de armazenamento, entre outros [NIST, 2011]. Este modelo possui o seguinte conjunto de características:

Recursos a pedido

A estrutura da *Cloud* é capaz de rapidamente reservar recursos, como capacidade de processamento e de armazenamento, consoante a necessidade das aplicações sem intervenção humana [NIST, 2011]. Por exemplo, se um *website* está alojado na *Cloud* e num determinado instante o número de visualizadores aumenta, então os recursos alocados também devem aumentar para que os clientes não sintam uma quebra de desempenho. Por outro lado, se o número de visualizadores do *website* diminui, os recursos devem ser libertados para que não consumam recursos desnecessariamente. Esta reserva e libertação de recursos deve ser automática para que o responsável do *website* não o tenha de monitorizar continuamente e indicar que são necessários mais ou menos recursos.

Acesso Oblíquo

Os recursos disponibilizados devem ser acedidos através de protocolos *standard* e em diferentes tipos de plataformas, por exemplo, computadores, *smarphones* e *tablets* [NIST, 2011].

Partilha dos recursos

Os recursos disponibilizados na *Cloud* são partilhados pelos vários consumidores [NIST, 2011], isto é, os consumidores alugam recursos, não os compram. Por exemplo, um consumidor que possui um *website* na *Cloud* não compra um disco com uma determinada capacidade, mas aluga uma determinada quantidade de memória de forma a satisfazer as suas necessidades, sendo que o mesmo disco pode estar a ser utilizado por vários consumidores em simultâneo.

Elasticidade

O sistema permite uma rápida e fácil escalabilidade, aparentando para os consumidores que a quantidade de recursos é ilimitada [NIST, 2011].

Monitorização dos Recursos

A *Cloud* deve controlar e otimizar automaticamente os recursos a partir de métricas apropriadas, como por exemplo memória consumida, tempo de processamento, largura de banda requerida. Esta monitorização pode ser efetuada tanto pelos fornecedores, como pelos consumidores [NIST, 2011].

Para além de respeitar estas características *Cloud Computing* possui três modelos de serviços: SaaS, PaaS, IaaS.

Software as a Service (SaaS)

Este modelo consiste em disponibilizar uma aplicação que é executada na estrutura da *Cloud*. Normalmente é acedida através de um *browser*, não sendo necessária qualquer instalação e configuração por parte do utilizador [NIST, 2011]. Um exemplo deste tipo de serviço é o "*Google Docs*", e consiste num *office online* (com editor de texto, folha de cálculo, entre outros), sendo apenas necessária uma conta para aceder à aplicação.

Platform as a Service (PaaS)

Este modelo disponibiliza um ambiente de desenvolvimento na *Cloud*, que permite aos utilizadores desenvolverem e executarem as suas próprias aplicações, mas sem terem de se preocupar com configurações de camadas de mais baixo nível como o sistema operativo, armazenamento, entre outros [NIST, 2011]. Basicamente disponibilizam uma máquina virtual para os utilizadores, mas com algumas restrições de acesso às configurações do sistema.

Infrastructure as a Service (IaaS)

Este modelo é idêntico ao PaaS, com a diferença de que os utilizadores têm acesso à camada mais inferior das máquinas virtuais, podendo configurar o sistema operativo, ou instalar outro, selecionar o método de armazenamento, entre outros [NIST, 2011].

Também foram definidas três formas de instalar um sistema de *Cloud Computing:* privado, público e híbrido.

Nuvens Privadas

As nuvens privadas encontram-se dentro de uma infraestrutura privada sob o controlo de uma determinada organização que gere as aplicações em execução. A organização que controla não necessita de possuir obrigatoriamente a infraestrutura onde se encontra a *Cloud*, podendo esta pertencer a uma terceira parte, ou mesmo uma combinação de ambas [NIST, 2011].

Nuvens Públicas

As nuvens públicas são utilizadas pelo público geral e normalmente encontram-se numa infraestrutura de uma organização pública [NIST, 2011].

Nuvens Híbridas

As nuvens híbridas são a combinação de várias nuvens (públicas e privadas) que apesar de serem independentes, possuem protocolos entre si que permitem a portabilidade de dados e aplicações [NIST, 2011].

2.3.2. Principais fornecedores

Desde o aparecimento da *Cloud Computing*, foram várias as instituições que tentaram aderir a esta ideia de negócio, umas com mais sucesso que outras. Como era de se esperar, grandes nomes da informática também tentaram aderir, como é o caso da Microsoft, Google, IBM, HP, entre outros. Mas não foram apenas gigantes da área, também outras instituições mais "pequenas" tentaram a sua sorte, e até conseguiram ter mais sucesso que os anteriores, ajudando assim a aumentar a sua relevância no mercado. Exemplos disso são a Amazon, Rackspace, Salesforce e Joyente. É preciso ter em consideração que existem outras instituições com uma boa cota de mercado na área de *Cloud Computing*, mas apenas serão abordadas algumas das instituições.

Com tantos fornecedores de serviços da *Cloud Computing*, qual escolher? Qual o melhor? Na tentativa de responder a estas questões, um conjunto de investigadores desenvolveram o sistema CloudCmp [Li, Yang, Kandula, & Zhang, 2010] que compara os vários fornecedores. Nessa comparação são tidos em apreciação vários aspetos, tais como quais as linguagens de programação disponibilizadas, suporte técnico, a elasticidade do sistema, o tipo e eficiência de armazenamento, entre outros.

A conclusão foi que não existe um fornecedor melhor que todos os outros em todos os aspetos, apenas que uns fornecedores são mais indicados para um tipo de modelo de serviço (SaaS, PaaS, IaaS) do que outros, e até dentro do mesmo modelo, dependendo das aplicações dos clientes, pode ser mais indicado um fornecedor. Por exemplo, usando os recursos mais básicos disponibilizados pelo C1 e o C2 foi possível concluir que apesar de o C1 ser 30% mais caro, pode terminar tarefas duas vezes mais rápido que o C2. O C1 e o C2 correspondem a dois fornecedores de *Cloud Computing*, mas para o foco das conclusões a que chegaram ser os resultados obtidos invés da classificação dos fornecedores, a CloudCmp disponibiliza os resultados associados a códigos: C1, C2, C3 e C4 invés de utilizar os nomes dos fornecedores [Li, Yang, Kandula, & Zhang, 2010].

Outro exemplo é que existem atualmente dois modelos de preços no que diz respeito a operações de armazenamento. O primeiro modelo é basear o custo de uma operação no número

de ciclos do CPU necessários para a executar, ou seja, quanto mais complexa for a operação mais caro é executá-la. O outro modelo é fixar o custo da execução de uma operação independentemente da sua complexidade. Os fornecedores servem-se destes modelos. O cliente tem de ter isto em conta na altura de escolher o fornecedor [Li, Yang, Kandula, & Zhang, 2010].

3. Ambiente de execução distribuída de aplicações

3.1. Organização da MVP

A MVP é constituída por um conjunto de nós de processamento independentes que cooperam entre si para disponibilizar os seus recursos de forma a aparentarem pertencer a um único nó. Como tal, um dos nós de processamento é responsável por interagir com os clientes, escondendo os restantes, aparentando assim que todos os recursos lhe pertencem. O nó que desempenhar essa função é designado de *Front End*.

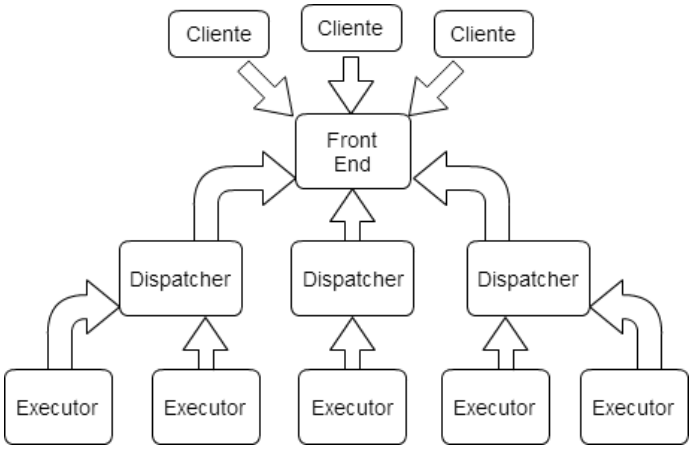

Figura 3.1. Arquitetura em árvore da MVP

Quando o *Front End* recebe o pedido de execução de uma determinada tarefa, coloca-a numa fila de espera até que um dos restantes nós da rede esteja disponível para processá-la. Esse nó passa a ser designado por *Dispatcher* e a sua função é obter as subtarefas da tarefa, através da aplicação do modelo de paralelização especificado no capítulo 1. Sempre que existir uma subtarefa disponível para ser processada e um nó de processamento disponível para o fazer, o *Dispatcher* envia a subtarefa para esse nó que começa a desempenhar a função de *Executor*. Este recebe a subtarefa, processa-a e notifica o *Dispatcher* que terminou. Após notificar o *Dispatcher*, o *Executor* volta a ser um simples nó, sem função atribuída, a aguardar que seja selecionado novamente para desempenhar uma função. Quando todas as subtarefas forem processadas, o *Dispatcher* notifica o *Front End* que já terminou, deixando assim de desempenhar a função de *Dispatcher*, tal como acontece com um *Executor*. Durante todo este processo o cliente pode pedir ao *Front End* o estado da tarefa, ao qual este responde:

- EM ESPERA se a aplicação ainda não foi atribuída a nenhum *Dispatcher*;
- EM EXECUÇÃO se a aplicação já está atribuída a um *Dispatcher*;
- EXECUTADA se o resultado já está disponível.

3.1.1. Ciclo de vida geral

Já se sabe que a MVP é constituída por um grupo de nós, que vão desempenhar diferentes papéis (*Front End*, *Dispatcher* e *Executor*) ao longo do tempo. Mas como um nó de processamento sabe que função desempenhar?

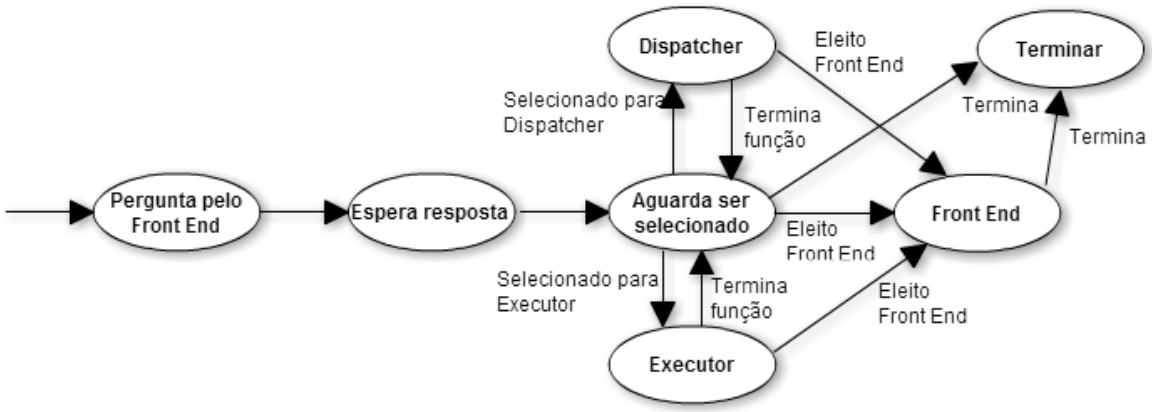

Figura 3.2 Ciclo de vida geral de um nó

Quando um nó de processamento é lançado, o seu primeiro objetivo é descobrir que nó desempenha a função de *Front End*, sendo para isso enviada para os restantes nós uma mensagem a questionar que nó desempenha essa função. Após enviar todas as mensagens, o nó aguarda pela resposta do *Front End* durante um determinado período de tempo. Quer tenha sido recebida a resposta, quer o tempo de espera tenha expirado, o nó passa para um estado de espera onde aguarda que lhe seja atribuída uma função a desempenhar. A diferença é que se o nó recebeu a resposta do *Front End*, significa que o *Front End* tem conhecimento de que agora o nó pertence ao agregado e está disponível para desempenhar uma função, e o nó tem conhecimento de que existe um *Front End* para interagir com os clientes. Mas se não receber uma resposta, enquanto o nó aguarda que lhe seja atribuída uma função, ocorre uma eleição para determinar qual o nó que desempenhará a função de *Front End*.

O esquema da figura anterior não faz referência a uma eleição, porque o objetivo desta secção não é detalhar todas as fases do ciclo de vida mas dar uma noção de alto nível de como os nós alteram de funções na MVP. Os detalhes surgirão nas secções seguintes.

Quando um nó é eleito para *Front End* nunca mais altera de estado. Por outro lado, se um nó for selecionado para *Dispatcher* ou *Executor*, uma vez a função ter sido terminada, o nó volta a aguardar por ser selecionado para desempenhar uma função.

Quando um nó pretende deixar de fazer parte da MVP envia uma mensagem a avisar o *Front End* que já não está disponível, para que seja selecionado outro nó para a função que desempenhava caso seja esse o caso. Se o nó que pretende sair da MVP for o próprio *Front End,*

então envia uma mensagem a todos os nós a avisar, dando assim início a uma nova eleição. O vencedor passa a ser o novo *Front End*, os restantes continuam a desempenhar as funções que lhes estavam destinadas antes da eleição, *Dispatcher*, *Executor* ou a aguardar.

3.1.2. Ciclo de vida de um *Front End*

A principal função do *Front End* é aguardar por pedidos dos clientes, quando este é a execução de uma tarefa seleciona um dos outros nós para desempenhar a função de *Dispatcher*. De seguida, envia uma mensagem a notificar o nó de que irá passar a desempenhar a função de *Dispatcher* e aguarda por uma resposta por um período limitado. Se o nó não responder, assume-se que este não se encontra disponível e volta a selecionar outro nó, a enviar a notificação, e a aguardar pela resposta, até um responder que recebeu a notificação. Se não existirem nós, nesse caso o *Front End* aguarda até que novos nós se juntem ao agregado, ou que nós já existentes passem a estar disponíveis para desempenhar uma função. Quando um nó responde, o *Front End* envia-lhe a tarefa e repete o ciclo.

Figura 3.3 Ciclo de vida do *Front End*

3.1.3. Ciclo de vida de um *Dispatcher*

Quando um nó recebe a mensagem do *Front End* a indicar que passará a ser um *Dispatcher*, responde a indicar que se encontra disponível e aguarda que o *Front End* envie a tarefa a distribuir. Depois de a receber, enquanto existirem subtarefas prontas para serem processadas, seleciona uma delas, seleciona um nó que esteja a aguardar por ser selecionado e envia uma mensagem a notificá-lo. Depois de aguardar um determinado tempo se não tiver recebido uma mensagem desse nó a indicar que se encontra disponível, seleciona outro até algum

responder. Nesse caso, o *Dispatcher* envia a subtarefa para o *Executor* e verifica novamente se existem subtarefas para serem processadas. Caso não exista, aguarda pelos resultados das subtarefas já distribuídas até que uma nova subtarefa fique disponível. Quando existirem subtarefas prontas a serem processadas, seleciona uma tarefa, e um nó para processá-la. Quando todas as subtarefas tiverem sido processadas, o *Dispatcher* notifica o *Front End* e volta a aguardar que lhe seja atribuída alguma função.

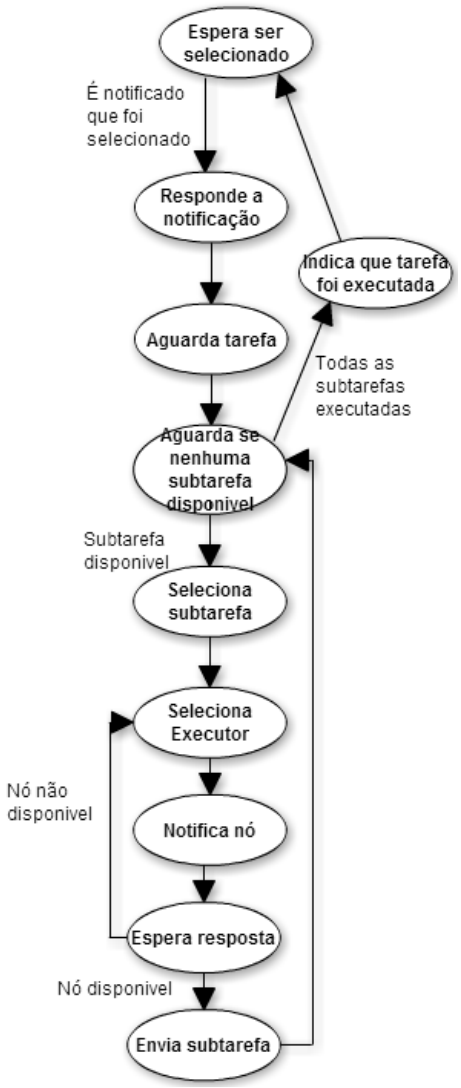

Figura 3.4 Ciclo de vida de um *Dispatcher*

3.1.4. Ciclo de vida de um *Executor*

Quando um nó recebe uma mensagem de um *Dispatcher* a indicar que passará a desempenhar a função de *Executor*, envia uma confirmação e aguarda que o *Dispatcher* lhe envie

a subtarefa a processar. Depois de processar a subtarefa notifica o *Dispatcher* e volta a aguardar por uma função para desempenhar.

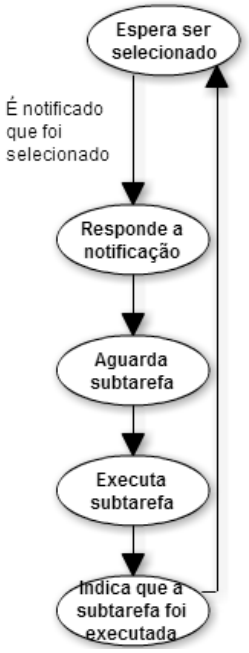

Figura 3.5 Ciclo de vida de um *Executor*

3.2. Organização interna de um nó

A arquitetura do nó deve permitir a execução do ciclo de vida demonstrado anteriormente de uma forma eficiente, mas também ser flexível o suficiente para permitir adicionar novas funcionalidades ou alterar funcionalidades existentes, não implicando refazê-la por completo. Para além das funcionalidades estabelecidas, é necessário que a arquitetura suporte vários protocolos responsáveis por gerir agregado de nós, como por exemplo, deteção das falhas dos nós, recuperar dessas falhas, manter os nós atualizados, entre outros. A comunicação entre os nós é efetuada através do paradigma de troca de mensagens.

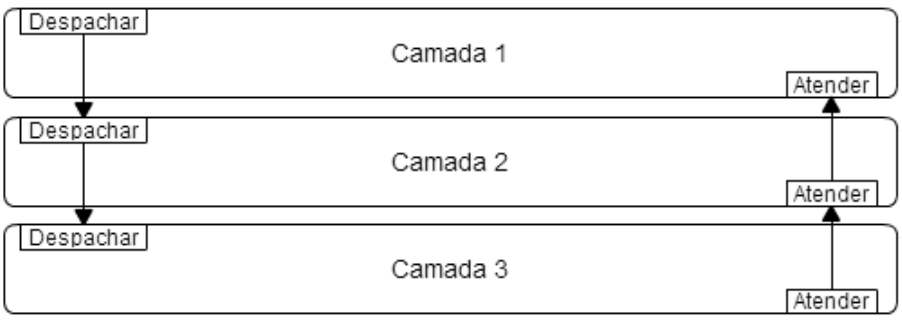

Figura 3.6. Sistema de camadas

A arquitetura de um nó encontra-se divida em camadas independentes que implementam as suas próprias funcionalidades. Cada camada apenas tem conhecimento da camada que se encontra imediatamente acima e abaixo da própria, de forma a reencaminhar as mensagens trocadas entre os nós, para uma das camadas adjacentes quando a mensagem não lhe é destinada. Para isso, as camadas implementam uma interface comum definida por duas funcionalidades: `atender` e `despachar`. Quando uma camada recebe uma mensagem pela função `atender` deve verificar se a mensagem se destina à própria. Caso se destine, deve processá-la, caso contrário deve passá-la à camada superior através da função `atender` dessa camada. Se a camada receber uma mensagem pela função `despachar`, então deve passar a mensagem para a camada inferior através da mesma função, exceto a camada inferior que deve enviar a mensagem para o nó a que se destina.

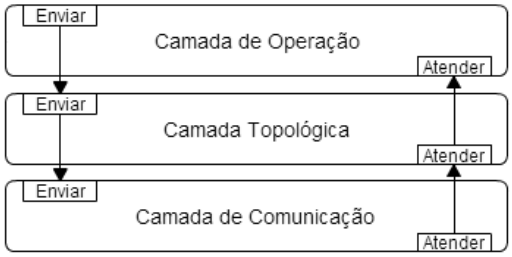

Figura 3.7. Camadas da arquitetura

As camadas que constituem a arquitetura do nó são: a **Camada de Comunicação**, a **Camada Topológica** e a **Camada de Operação**. A **Camada de Comunicação** é a camada inferior e é responsável por interligar os nós da MVP. A camada intermédia é a **Camada Topológica** e possui os protocolos responsáveis por gerir o agregado. A camada superior é a **Camada de Operação** e é a camada que desempenha o ciclo de vida de um nó.

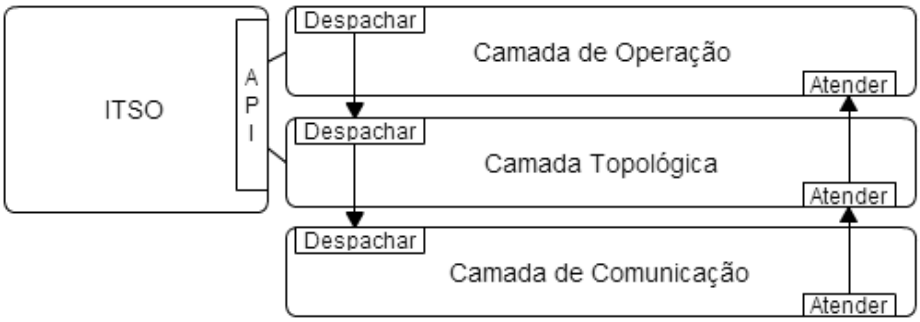

Figura 3.8. Camadas da arquitetura do nó com **ITSO**

As camadas possuem os protocolos e algoritmos referidos anteriormente, mas não possuem os dados que resultam ou são necessários para alimentar os mesmos. Por exemplo, a informação de quais os nós que se encontram ativos no agregado tanto pode ser utilizada por protocolos de gestão da **Camada Topológica**, como pode ser utilizada pela **Camada de Operação**. Portanto, essa informação não se pode encontrar no interior de uma determinada camada, mas sim numa zona externa a que todas possam aceder. A essa zona deu-se o nome de **Informação Topológica de Suporte às Operações** (**ITSO**) e possui todos os dados necessários para o funcionamento de um nó, disponibilizando uma API que é utilizada pelas camadas para aceder aos dados.

O **ITSO** não armazena toda a informação do sistema. Quando um nó é selecionado para desempenhar a função de *Executor* ou *Dispatcher*, recebe uma mensagem com a informação da localização da tarefa e subtarefas a processar, não contendo os ficheiros de código e dados. Para os obter é utilizado o componente **Sistema de Ficheiros Distribuído** (**SFD**).

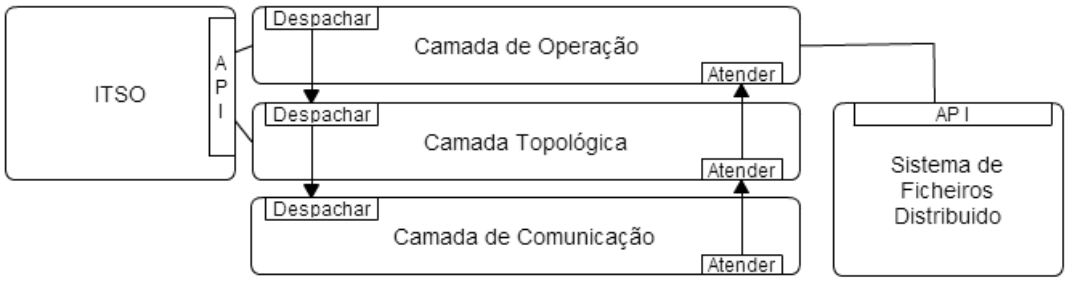

Figura 3.9 Arquitetura interna do nó

O **SFD** é o componente responsável por fazer o *download* e *upload* de ficheiros sendo disponibilizada uma API que é utilizada pela **Camada de Operação**, a única camada que processa as tarefas e subtarefas.

3.2.1. ITSO – Informação Topológica de Suporte às Operações

ITSO é o componente onde são armazenados os dados utilizados pelos vários protocolos e funcionalidades de um nó. Como um dos objetivos da arquitetura é ser flexível, para permitir futuras alterações quer nos protocolos, quer nas funcionalidades, sem refazer toda a arquitetura, este componente ao invés de possuir uma estrutura de dados única que armazena os dados de todos os protocolos, é constituído por um conjunto de componentes que armazenam separadamente os dados relativos a um determinado protocolo ou funcionalidade.

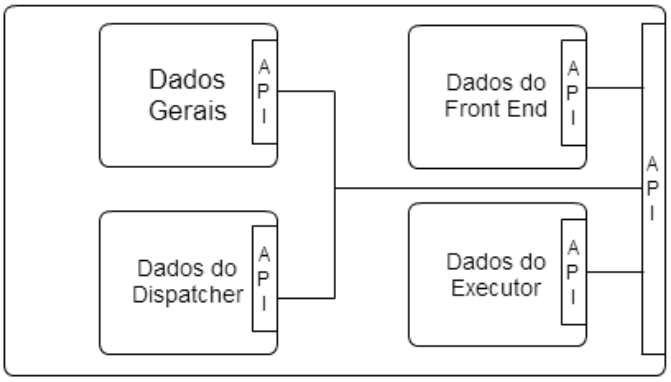

Figura 3.10. **ITSO**

Tal como demonstra a figura anterior, o **ITSO** é constituído por quatro subcomponentes: **Dados Gerais**, **Dados do *Front End***, **Dados do *Dispatcher*** e **Dados do *Executor***. O componente **Dados Gerais** suporta os protocolos da **Camada Topológica**, isto é, possui a informação base de cada nó que pertence ao agregado. Os restantes componentes suportam as funcionalidades da **Camada de Operação**, possuindo informações mais detalhadas dos nós mais relevantes do agregado. Cada um destes quatro componentes possui uma API de acesso aos dados que armazenam, sendo que as quatro APIs juntas formam a API do **ITSO**.

3.2.1.1. Dados Gerais

Tal como já foi referido, este componente tem como objetivo suportar as operações de gestão do agregado de nós, possuindo uma lista de nós que possivelmente pertencem ao agregado. Possivelmente porque nem todos os nós da lista podem estar ativos. Para além de identificar que nós se encontram ativos, é necessário identificar qual a função que desempenham, isto é, identificar que nó desempenha a função de *Front End*, que nós estão ocupados (desempenham a função de *Dispatcher* ou *Executor*) e que nós estão livres (aguardam que lhes seja atribuída uma função). Cada elemento da lista possui: um endereço IP, um porto de escuta, e um estado. Os dois primeiros identificam o nó, o estado pode ser um de cinco possíveis:

- INACTIVE – o nó não está ativo, ou seja, não pertence ao agregado;
- FRONT_END – o nó desempenha a função de *Front End*;
- DISPATCHER – o nó desempenha a função de *Dispatcher*;
- EXECUTOR – o nó desempenha a função de *Executor*;
- WAITING – o nó está ativo, mas não possui nenhuma função atribuída.

Operação	Descrição
getAllNodes()	obtém o endereço IP e porto de escuta de todos os nós que possivelmente pertencem à MVP
getActiveNodes()	obtém o endereço IP e porto de escuta de todos os nós que realmente pertencem à MVP
getNodesWithState(state)	obtém o endereço IP e porto de escuta de todos os nós que possuem um determinado estado
getState(ip, port)	obtém o estado atual do nó com um determinado endereço IP e porto de escuta
setState(ip, port, state)	define o estado do nó com um determinado endereço IP e porto de escuta

Tabela 3.1 API de acesso do componente **Dados Gerais**

Após a definição das estruturas de dados e da API de acesso a estes do componente **Dados Gerais**, surgiu uma questão: qual o método mais eficiente para os protocolos das camadas detetarem a ocorrência de uma alteração dos dados do **ITSO**? Foram encontradas duas soluções para este desafio, ou através de *pooling*, ou através de um mecanismo de subscrição.

A primeira solução consiste nos protocolos das camadas consultarem periodicamente a informação armazenada no **ITSO** à procura de alterações, o que provoca dois problemas. Primeiro, mesmo que não ocorra nenhuma alteração nos dados os protocolos irão consumir os recursos da máquina a consultá-los periodicamente. Segundo, mesmo que ocorram alterações nos dados, no pior dos casos essas alterações só irão ser detetadas após um ciclo de consulta completo.

A segunda solução é o oposto da primeira, na perspetiva de que não são os protocolos que consultam as estruturas de dados do **ITSO**, mas é o próprio que notifica os protocolos das camadas de que ocorreu uma alteração dos dados, e ainda pode indicar que dados foram alterados, eliminando assim a necessidade de os protocolos os consultarem. Comparando com a primeira solução, não são desperdiçados recursos periodicamente a consultar os dados, e quando ocorrem alterações nestes, os protocolos são notificados imediatamente. Para implementar este mecanismo foi necessário adicionar uma lista de subscritores a cada um dos subcomponentes do **ITSO**. Um subscritor é um componente que implementa um conjunto de operações específicas, no caso dos **Dados Gerais** os componentes que desejem ser seus subscritores devem implementar as operações:

- nodeChange(ip, port, newState);
- stateChange(newState);
- frontEndChange(newIP, newPort);

Quando ocorrer uma alteração nos dados, os componentes que são subscritores dos **Dados Gerais** serão notificados através da execução de, pelo menos, uma das operações anteriores: `nodeChange` quando um nó altera o seu estado, independentemente do nó e do estado para que foi alterado, `stateChange` quando o estado do próprio nó é alterado, e `frontEndChange` quando o nó que desempenha a função de *Front End* é alterado.

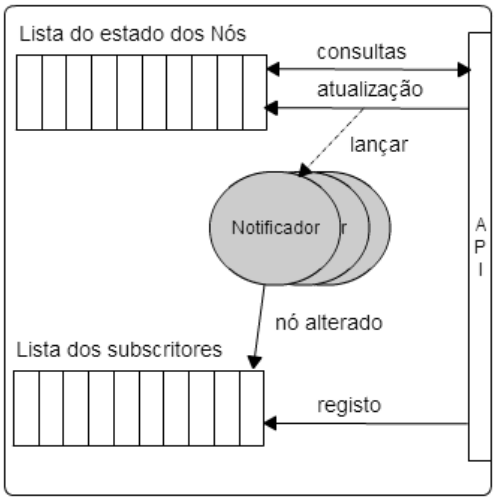

Figura 3.11. **Dados Gerais**

Como é possível verificar pela figura anterior, quando ocorre uma alteração na lista do estado dos nós é lançada uma entidade ativa por cada subscritor registado. A entidade ativa é denominada de `Notificador`, e será responsável pela invocação da operação associada à alteração de um único subscritor. Desta forma os subscritores são notificados concorrentemente.

Também é possível verificar que existe uma nova operação na API deste componente, a operação registo. É através desta operação que os subscritores são adicionados à lista para serem notificados.

3.2.1.2. Dados do *Front End*

O objetivo deste componente é suportar as operações dum *Front End*. A principal tarefa é receber as tarefas para execução dos clientes, atribuí-las a um *Dispatcher*, e guardar os resultados até que os clientes os requisitem. Para suportar estas funcionalidades, este componente possui três filas de espera com diferentes comportamentos.

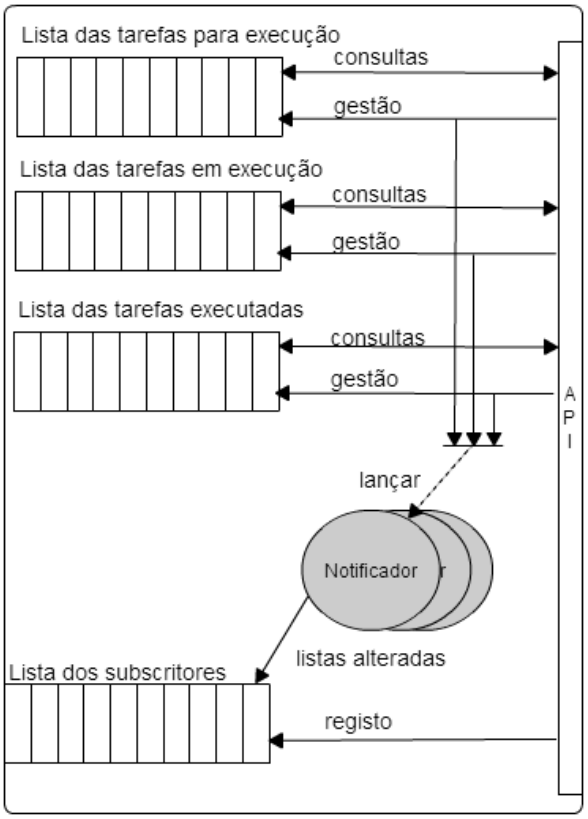

Figura 3.12. **Dados do *Front End***

Uma das filas de espera armazena as tarefas a aguardar a sua execução, e possui duas operações: `addTaskToExecute` e `waitTask`. A primeira operação simplesmente adiciona uma nova tarefa ao fim da fila de espera, a operação `waitTask` bloqueia a entidade ativa que a chamar enquanto a fila não possuir nenhuma tarefa.

A segunda fila de espera possui as tarefas em execução, associando a cada uma o nó a que foi atribuída (o nó *Dispatcher* encarregado de a distribuir). Existem três operações que atuam sobre esta fila: `setTaskInExecution`, `taskExecuted` e `taskNotExecuted`. A primeira operação retira a primeira tarefa da fila anterior, adiciona-a a esta fila associada a um nó, e retorna a referência da tarefa. As últimas duas operações retiram uma determinada tarefa da lista e colocam-na ou na primeira fila, no caso da operação `taskNotExecuted` (normalmente quando ocorre um erro e a tarefa não pode ser executada), ou na terceira fila de espera, no caso da operação `taskExecuted`.

A terceira fila possui as tarefas que foram executadas e que aguardam que os clientes obtenham os resultados. Sobre esta fila atuam duas operações: `taskExecuted` e `destroyTask`. A primeira já foi referida, apenas adiciona uma nova tarefa à lista, enquanto a segunda remove a tarefa da lista, normalmente após os clientes obterem os resultados.

Para além das operações anteriores, existe a operação `taskState` que indica o estado da tarefa (esperando, executando, executada) consoante a fila de espera em que se encontra, e a operação de registo de subscritores, porque tal como no componente **Dados Gerais** este componente possui um sistema de subscrição. Portanto, quando alguma destas três filas é alterada, isto é, é adicionada ou removida alguma tarefa de algumas destas três filas, os subscritores são notificados através das operações seguintes.

Operação	Descrição
newTaskToExecute(task)	quando uma nova tarefa é adicionada à fila de tarefas para execução
taskExecuting(task, dispatcher)	quando uma tarefa é adicionada à fila de tarefas em execução
taskExecuted(task, dispatcher)	quando uma tarefa é retirada da fila de tarefas em execução e adicionada à fila de tarefas executadas
taskNotExecuted(dispatcher)	quando uma tarefa é retirada da fila de tarefas em execução e adicionada à fila de tarefas para execução
taskDestroyed(taskID)	quando uma tarefa é retirada da fila de tarefas executadas

Tabela 3.2 Métodos que um subscritor deve implementar

3.2.1.3. Dados do *Dispatcher*

À semelhança do componente anterior, o objetivo deste componente é suportar as operações do nó, mas para a função de *Dispatcher*. A principal função é distribuir as subtarefas de uma tarefa por *Executors* e, depois de todas as subtarefas serem executadas, notificar o *Front End*. Para suportar estas operações é utilizada uma estrutura para armazenar a tarefa a distribuir e uma lista onde são colocados os nós que se encontram a executar uma subtarefa (os *Executors*).

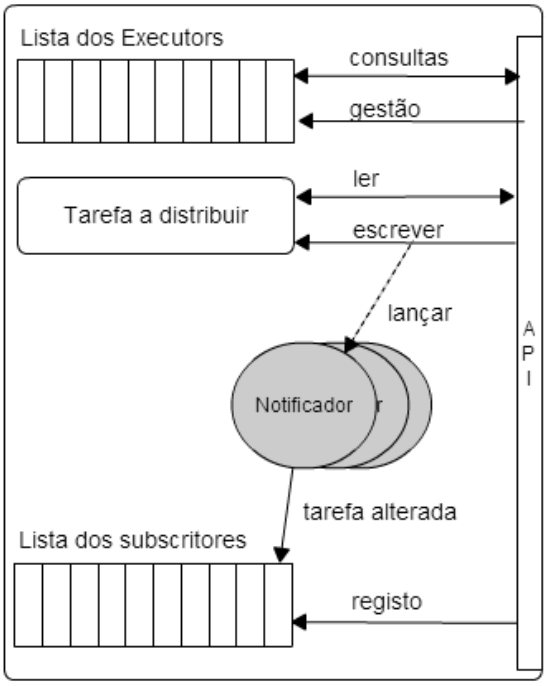

Figura 3.13. **Dados do *Dispatcher***

Tal como nos componentes anteriores, **Dados do *Dispatcher*** possui um sistema de subscrição, onde são notificados quando a tarefa é alterada através da operação `taskChange` que o subscritor deve implementar. A estrutura que armazena os dados de uma tarefa é constituída por os seguintes campos.

Campo	Descrição
ID	identifica unicamente a tarefa
RepositoryHost	indica o endereço IP do repositório de ficheiros
RepositoryPort	indica o porto do repositório de ficheiros
Directory	indica o diretório que contém os ficheiros da tarefa
File	indica o nome do ficheiro que descreve as subtarefas e como estas se encontram organizadas

Tabela 3.3 Estrutura de dados de suporte a uma tarefa

```
<subtasks>
  <subtask>
    <command>
        <jarfile>file.jar</jarfile >
        <classname>package.class</classname>
        <param>source.txt</param>
                ...
        <param>result.txt</param>
    </command>
    <jarfile>file i.jar</jarfile >
    <entryfile>source.txt</entryfile >
                ....
    <entryfile>sourceX.txt</entryfile >
    <resultfile>res.txt</result>
                ....
    <resultfile>resX.txt</result>
  </subtask>
</subtasks>
```

Figura 3.14 Ficheiro de descrição

O ficheiro que descreve as subtarefas é um ficheiro XML e respeita a estrutura demonstrada na figura anterior. O elemento raiz é denominado de `subtasks`, e possui uma lista de elementos `subtask`. O elemento `subtask` possui quatro tipos de elementos: `command`, `jarfile`, `entryfile` e `resultfile`.

O elemento `command` é obrigatório e só deve existir um. É constituído por três tipos de elementos: `jarfile`, `classname` e `param`. O elemento `jarfile` é obrigatório, único e identifica o ficheiro que possui o código da subtarefa. O elemento `classname` também é obrigatório e único, e indica o pacote e a classe que possui a função *main* do código. O elemento `param` não é obrigatório e podem existir vários, sendo utilizados como parâmetros de entrada da função *main*.

Os restantes elementos da `subtask` indicam os nomes dos ficheiros necessários para a execução da subtarefa. O elemento `jarfile` indica o ficheiro com o código a executar, é obrigatório e único. O elemento `entryfile` indica os ficheiros com os dados de entrada para a execução da tarefa. O elemento `resultfile` indica os ficheiros com os resultados da execução da tarefa.

3.2.1.4. Dados do *Executor*

O último componente do **ITSO** é o **Dados do *Executor*** que é responsável por suportar as operações de um *Executor*, receber uma subtarefa, executá-la e informar o *Dispatcher* que já terminou. Este componente possui a informação de que nó desempenha a função de *Dispatcher* e uma estrutura que armazena os dados de uma subtarefa.

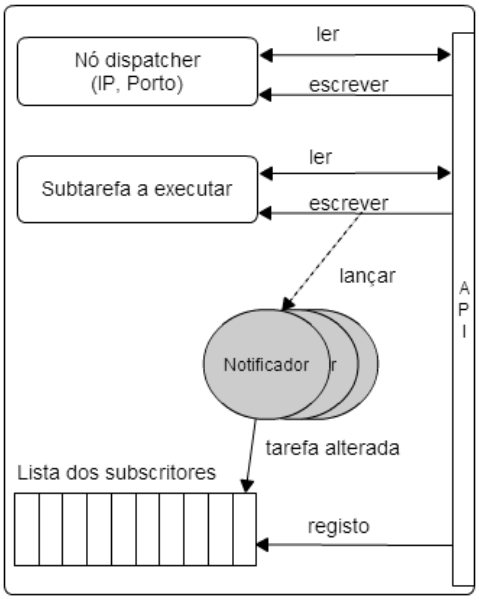

Figura 3.15. **Dados do *Executor***

Este componente também possui um sistema de subscritores que são notificados quando a subtarefa armazenada é alterada, através da operação `subtaskChange` que estes implementam. A estrutura de dados que armazena a informação de uma subtarefa é constituída pelos seguintes campos.

Campo	Descrição
ID	identifica unicamente a subtarefa dentro da tarefa
RepositoryHost	indica o endereço IP do repositório de ficheiros
RepositoryPort	indica o porto do repositório de ficheiros
Directory	indica o diretório que contém os ficheiros da tarefa
JarFile	indica o nome do ficheiro com extensão jar que contém o código da subtarefa
ClassName	Indica o nome da classe que possui a função *main*
EntryFiles	indica o nome dos ficheiros com os dados de entrada
ResultFiles	indica o nome dos ficheiros com os resultados
Parameters	indica os parâmetros a passar à função *main*

Tabela 3.4 Estrutura de dados de suporte a uma subtarefa

3.2.2. Camada de Comunicação

Esta camada é responsável pela comunicação entre os nós da MVP, como tal é a camada responsável por enviar e receber mensagens, através do protocolo TCP/IP.

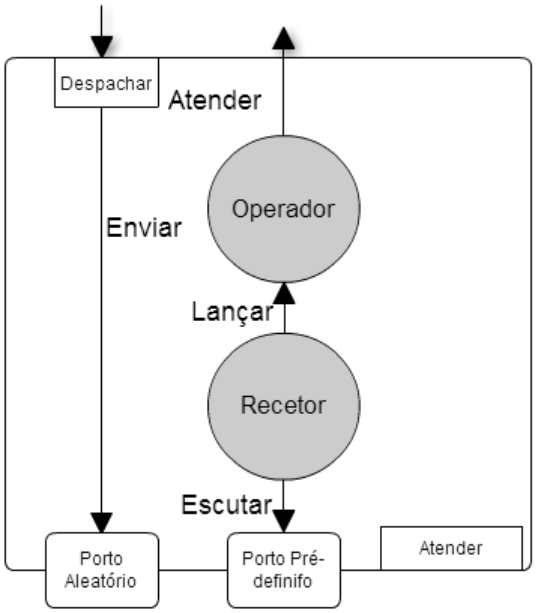

Figura 3.16. **Camada de Comunicação**

Esta camada implementa um servidor e um cliente de mensagens sendo apenas constituído por dois tipos de entidades ativas, o `Recetor` e o `Operador`, e dois portos. Sempre que uma mensagem é recebida proveniente da camada superior, através da operação `despachar`, esta camada envia através de uma porto aleatório a mensagem para o nó a que se destina, atuando assim como cliente. Por outro lado, sempre que uma mensagem é recebida pelo porto pré-definido, o `Recetor` é acordado, lançando de seguida um novo `Operador` e volta a bloquear. O `Operador` é a entidade responsável por processar a mensagem, passando-a à camada superior através da operação `atender`. Desta forma é possível processar múltiplas mensagens em concorrência, já que cada mensagem é processada por um `Operador` diferente.

3.2.3. Camada Topológica

Esta camada possui os protocolos responsáveis pela gestão do agregado de nós. Mais concretamente possui quatro protocolos: eleição do *Front End,* manter a coerência dos dados (atualização), deteção de falhas e recuperação das mesmas.

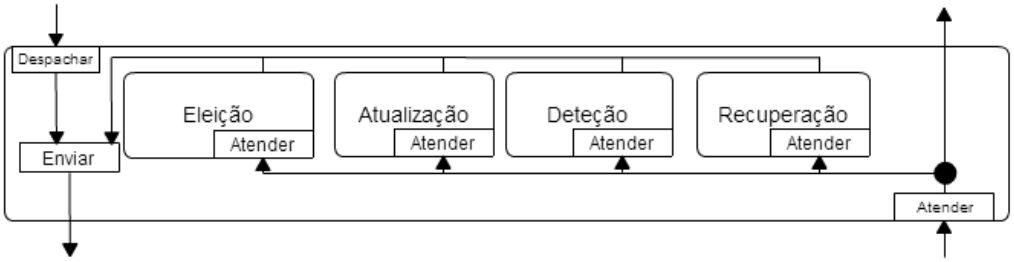

Figura 3.17. **Camada Topológica**

Cada um destes protocolos é independente dos restantes e possui um módulo responsável por implementá-lo. Sempre que esta camada recebe uma mensagem proveniente da camada superior (**operação**), despacha-a para a camada inferior (**comunicação**). Quando recebe uma mensagem proveniente da camada inferior verifica se esta se destina a algum dos seus módulos, e se for esse o caso, encaminha-a para o módulo correspondente, caso contrário reencaminha-a para a camada superior.

3.2.3.1. Módulo Eleição

O módulo **Eleição** é responsável por encapsular o algoritmo utilizado na seleção do nó que desempenhará a função de *Front End*. O algoritmo de eleição implementado é uma variante do *Bully* [Coulouris, Dollimore, & Kindberg, 2003]: ganha a eleição o nó com mais prioridade, sendo que esta depende do seu estado e ID. O estado mais prioritário é WAITING, seguido de EXECUTOR, e por último DISPATCHER. A razão desta ordenação deve-se ao facto de ser preferível nomear um nó que não desempenha qualquer função, do que nomear um nó que já desempenha uma função, de forma a introduzir o menor impacto possível na estrutura. No entanto, se no estado mais prioritário existir mais do que um nó, é necessário comparar o ID de forma a determinar qual tem mais prioridade. O ID é o valor produzido por uma função *hashing* do endereço IP e porto de escuta dos nós, sendo mais prioritário o maior.

Este módulo é um subscritor do componente **Dados Gerais** do **ITSO**, e como tal, é notificado sempre que o estado de um nó é alterado. Quando tal acontece relativamente ao nó que desempenhava até aqui a função de *Front End*, dá-se início a uma eleição.

Internamente, este módulo possui um registo para armazenamento da fase em que se encontra o processo de eleição. As fases possíveis são:

- `candidato` - o nó é candidato a ser o próximo *Front End;*
- `aguarda confirmação` - o nó sabe que não irá ganhar a eleição, e aguarda a indicação do vencedor;
- `perdedor` - o nó recebeu a indicação de quem é o vencedor;
- `vencedor` – o nó é o novo *Front End*.

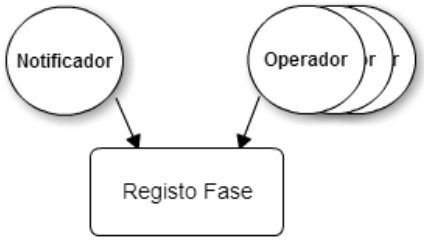

Figura 3.18 Entidades que atuam sobre o registo `Fase`

Sobre este registo atuam duas entidades ativas: o `Notificador` e múltiplas instâncias do `Operador`. O primeiro executa o processo de eleição, enquanto o segundo lida com as mensagens recebidas durante o mesmo.

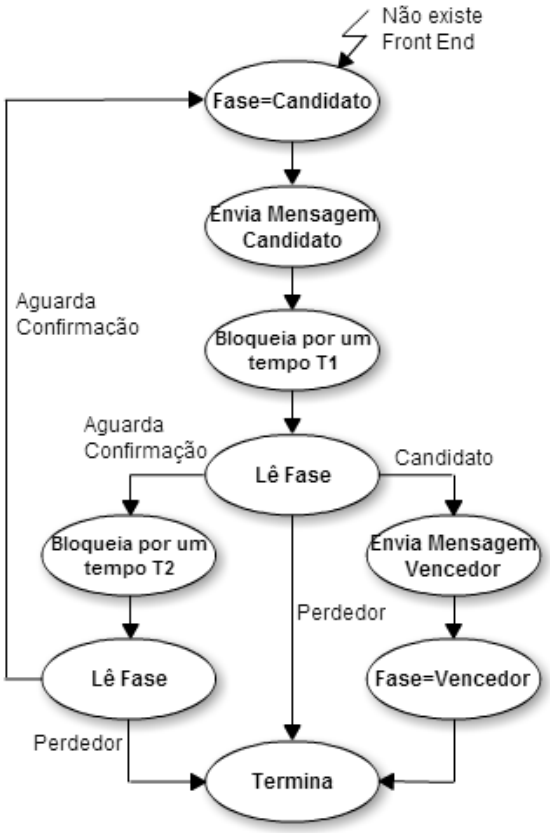

Figura 3.19 Ciclo de vida do algoritmo de eleição (`Notificador`)

Primeiramente, o `Notificador` escreve no registo do módulo a fase `candidato`. De seguida, envia para todos os nós que pertencem à MVP (incluindo os inativos) a mensagem CANDIDATO que contém o seu ID e estado. Posteriormente, a entidade bloqueia por um período de tempo T1. Após acordar lê o registo, e consoante o valor lido desempenha uma ação:

- `candidato` - envia a mensagem VENCEDOR para todos os nós que possivelmente pertencem à MVP, e escreve no registo a fase `vencedor`;
- `perdedor` - simplesmente termina;
- `aguarda confirmação` - bloqueia durante um período de tempo T2. Após acordar torna a ler o registo do módulo, caso o valor lido seja `perdedor` termina o ciclo, caso permaneça `aguarda confirmação` reinicia o processo de eleição.

As entidades `Operador` lidam com três tipos de mensagem: CANDIDATO (um nó anuncia que é candidato a desempenhar a função de *Front End*), VENCEDOR (um nó anuncia que é o novo *Front End*), e ABORTAR (um nó informa que o processo de eleição de ser abortado).

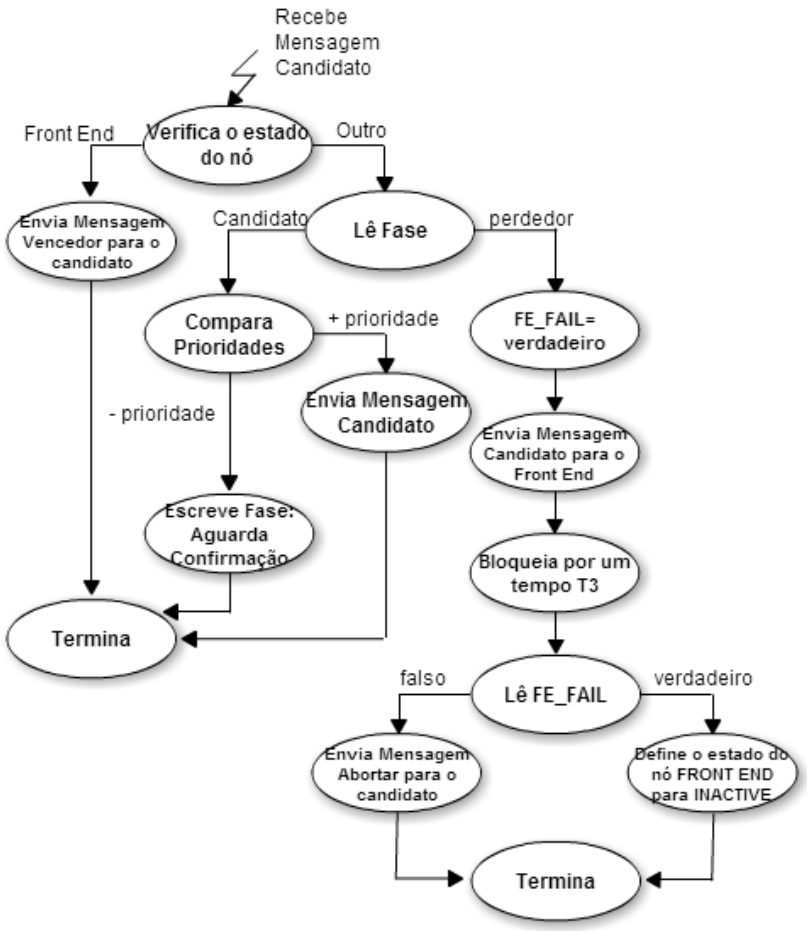

Figura 3.20 Ciclo de vida de um `Operador` quando recebe uma mensagem CANDIDATO

Quando um `Operador` recebe uma mensagem CANDIDATO verifica se o próprio nó desempenha a função de *Front End,* o que normalmente não deverá acontecer, mas pode ocorrer em casos de erro na transmissão. Caso tal se verifique, responde com a mensagem VENCEDOR para informar o candidato que o nó já desempenha a função de *Front End*. Caso contrário, lê o registo do módulo que indica a fase em que se encontra o processo de eleição. Se o valor lido for

`candidato` significa que o próprio nó é um candidato e compara a sua prioridade com a do nó que envia a mensagem. Se o nó que recebeu a mensagem possuir uma menor prioridade, é escrito no registo o valor `aguarda confirmação`. Caso possua maior prioridade, responde com uma mensagem CANDIDATO de forma a confirmar que o outro nó reconhece que tem menor prioridade. Por outro lado, se o valor lido do registo fase for `perdedor` significa que o nó ainda não iniciou um processo de eleição, ou seja, tem conhecimento de que um nó desempenha a função de *Front End*. Nesta situação, escreve no registo auxiliar FE_FAIL o valor `verdadeiro` e envia a mensagem CANDIDATO para o *Front End*. Após bloquear por um período de tempo T3, lê o registo FE_FAIL e caso este permaneça com o valor `verdadeiro`, significa que o nó que desempenhava a função de *Front End* não respondeu. Nestas condições, assume que o *Front End* falhou e altera o estado desse nó para INACTIVE no **ITSO**. Caso o valor lido seja `falso`, significa que existe um nó a desempenhar a função de *Front End*, nesse caso é enviada a mensagem ABORTAR para o candidato, para que este cancele o processo de eleição.

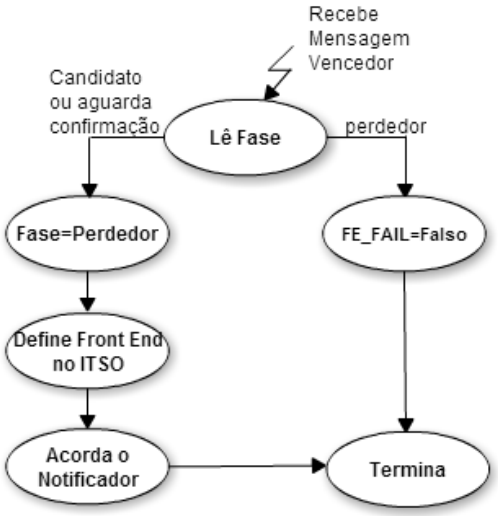

Figura 3.21 Ciclo de vida de um `Operador` quando recebe uma mensagem VENCEDOR

Se a mensagem recebida pelo nó for VENCEDOR é lido o registo que armazena a fase do processo de eleição. Se o valor lido for `perdedor` significa que a mensagem VENCEDOR foi enviada para confirmar que o nó que desempenha a função de *Front End* permanece ativo, portanto é escrito no registo FE_FAIL o valor `falso`. Caso o valor lido seja `candidato`, ou `aguardar confirmação`, significa que o nó perdeu a eleição, como tal escreve no registo que armazena a fase o valor `perdedor`, altera o estado do nó que enviou a mensagem para *FRONT END* no **ITSO**, e acorda o `Notificador` caso este se encontre bloqueado a aguardar a confirmação.

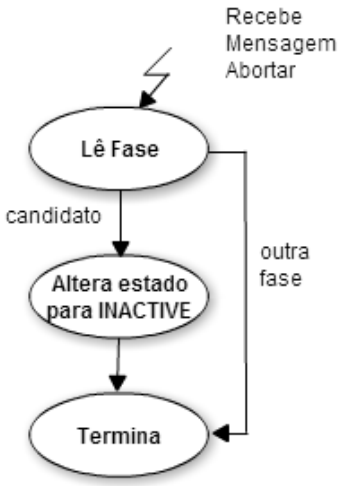

Figura 3.22 Ciclo de vida de um `Operador` quando recebe uma mensagem ABORTAR

Se a mensagem recebida for do tipo ABORTAR é lido o valor do registo que armazena a fase do processo de eleição. Se o valor lido for `candidato` significa que o nó iniciou um processo de eleição, mas já existe um nó a desempenhar a função de *Front End*. Nesta situação o nó altera o seu próprio estado para INACTIVE no **ITSO**, deixando de fazer parte do agregado. Se a fase for diferente de `candidato` significa que o nó iniciou um processo de eleição quando um nó já desempenhava a função de *Front End*, mas entretanto percebeu que este já existe e terminou o processo de eleição, não sendo necessário abandonar a MVP.

3.2.3.2. Módulo Atualização

Este módulo é responsável por manter a coerência dos dados entre os nós da MVP, mais concretamente os dados do componente **Dados Gerais** do **ITSO**. Para isso, este módulo é um subscritor desse componente sendo notificado sempre que um nó altera o estado. Essa alteração pode ter origem interna ou externamente. A alteração interna resulta de uma alteração efetuada pelo próprio nó através de outros módulos. Por exemplo, no módulo **Eleição** quando um nó é eleito *Front End*, ou módulo **Deteção** quando é detetada a falha de um nó. A alteração externa é desencadeada pela receção da mensagem UPDATE que contém o novo estado de um determinado nó.

Independentemente da origem da alteração do estado de um nó é lançado um `Notificador` no componente **Dados Gerais** responsável por notificar o módulo **Atualização**.

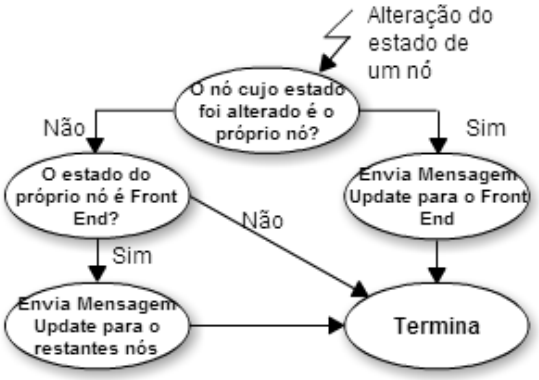

Figura 3.23 Ciclo de vida do `Notificador`

Em primeiro lugar, o `Notificador` verifica se o nó cujo estado foi alterado é o próprio nó. Caso tal se verifique, então envia uma mensagem UPDATE (com o seu novo estado) para o nó que desempenha a função de *Front End*. Caso contrário, se o estado do próprio nó seja FRONT_END, então envia uma mensagem UPDATE (com o novo estado) para os restantes nós da MVP.

Desta forma, quando um nó (que não seja *Front End*) sofrer uma alteração no seu estado notifica o *Front End*, que por sua vez notifica os restantes nós.

3.2.3.3. Módulo Deteção

Este módulo é responsável por garantir que quando um nó falha essa ocorrência é detetada, para isso, recorre-se a um mecanismo de *heartbeat.* Este mecanismo baseia-se na troca periódica de mensagens entre os nós permitindo que se mantenham informados sobre o estado de atividade dos restantes. Como esta mensagem é enviada periodicamente, a informação que ela contém é reduzida ao mínimo para evitar sobrecarregar o canal de comunicação. Neste caso, a mensagem trocada entre os nós é denominada por KEEP_ALIVE, apenas contém o endereço IP e o porto de escuta do nó que a envia.

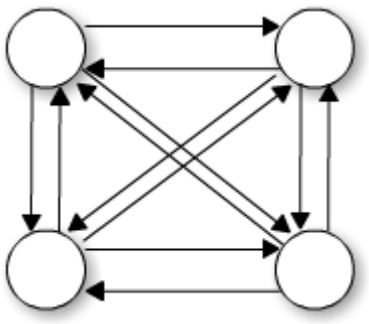

Figura 3.24 MVP constituída por 4 nós com troca de mensagem KEEP_ALIVE entre eles

Se a MVP for constituída por N nós e cada um desses nós enviar uma mensagem KEEP_ALIVE para os restantes, serão trocadas Nx(N-1) mensagens periodicamente. Seguindo o exemplo ilustrado pela figura anterior, se a MVP for constituída por 4 nós, serão trocadas periodicamente 12 mensagens KEEP_ALIVE.

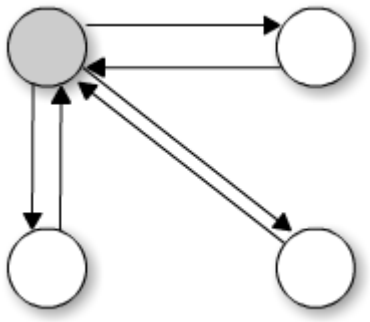

Figura 3.25 MVP constituída por 4 nós com troca de mensagem KEEP_ALIVE com o nó diferenciado

Na tentativa de reduzir o impacto que este protocolo provoca sobre a ocupação do canal de comunicação, reduziu-se o número de mensagens KEEP_ALIVE trocadas entre os nós. O método utilizado para esse efeito foi diferenciar um nó dos restantes, neste caso esse nó é o que desempenha a função de *Front End*. O *Front End* continua a enviar uma mensagem KEEP_ALIVE para os outros (N-1 mensagens), mas estes apenas enviam uma mensagem para ele (N-1 mensagens). Desta forma, o *Front End* deteta a falha de um dos nós restantes e cada um dos nós restantes deteta a falha do *Front End*. No primeiro caso, o *Front End* envia uma mensagem para os restantes a informá-los, através do módulo **Atualização**. Quando é o *Front End* a falhar, é eleito um novo nó para o seu lugar através do módulo **Eleição**.

Com este mecanismo são trocadas 2x(N-1) mensagens periodicamente entre os nós. Seguindo o exemplo ilustrado na figura anterior, sendo a MVP formada por 4 nós, apenas são trocadas 6 mensagens. Com 4 nós o número de mensagens trocadas reduziu-se em 50%, mas para um maior número de nós, por exemplo 1000, a redução do número de mensagens trocadas periodicamente é de 99%.

Para implementar este mecanismo, o módulo **Deteção** possui uma lista com os seguintes campos: endereço IP, porto de escuta e tempo. Os dois primeiros campos têm como objetivo identificar um nó, o campo tempo armazena um instante de tempo. Sempre que é recebida a mensagem KEEP_ALIVE, a entidade `Operador` associada verifica se o nó que a enviou já se encontra na lista (caso não se encontre é adicionado). De seguida, é atualizado o campo tempo com o instante de tempo atual (o instante em que foi recebida a mensagem KEEP_ALIVE).

Para além da lista, este módulo possui uma entidade ativa, o `Verificador`, que é acordado periodicamente para desempenhar duas tarefas. Em primeiro lugar, compara o campo tempo armazenado com o tempo atual para cada elemento da lista. Se a diferença for superior a um determinado limite, significa que não é recebida uma mensagem KEEP_ALIVE há mais tempo que o permitido, assumindo-se assim que o nó falhou. Nesse caso, o elemento é retirado da lista e o estado do nó associado é alterado para INACTIVE no **ITSO**. Em segundo lugar, o `Verificador` envia para os nós que permanecem na lista a mensagem KEEP_ALIVE.

Numa situação inicial a lista encontra-se vazia, então os nós nunca enviarão mensagens KEEP_ALIVE. Para solucionar este problema, tornou-se este módulo um subscritor do componente **Dados Gerais** do **ITSO**. Portanto, é notificado quando o estado de um nó é alterado, incluindo quando um nó passa a ser o *Front End.* Neste caso, é o `Notificador` responsável por este módulo a adicionar um elemento à lista com o endereço IP e porto do escuta do nó que desempenha a função de *Front End*, juntamente com o tempo atual. Desta forma, quando o `Verificador` acordar, a lista possui um único elemento, o *Front End*, e enviar-lhe-á a mensagem KEEP_ALIVE. Quando tal acontecer, o *Front End* receberá as mensagens KEEP_ALIVE dos restantes nós e adicioná-los-á à lista. Nessa altura, a lista do *Front End* possui os restantes nós, e a lista dos restantes nós possui o *Front End*.

3.2.3.4. Módulo Recuperação

Quando um nó falha é necessário que nenhuma informação imprescindível seja perdida. A solução deste problema implica o armazenamento de dados importantes noutros locais. Como será possível perceber nas secções seguintes, a única informação que, se perdida, não é possível recuperar, é a informação que se encontra no componente **Dados do *Front End*** do **ITSO**. Como tal, é necessário que outros nós possuam uma cópia deste componente, para caso o *Front End* falhar, o nó eleito possa recuperar informação.

Este módulo possui então duas responsabilidades: primeiro, garantir que outros nós possuam cópias atualizadas do componente **Dados do *Front End***; segundo, recuperar os dados, se necessário, sempre que um novo nó é eleito *Front End*. Para que tal seja possível, o módulo **Recuperação** contém uma lista que identifica os nós possuidores de uma cópia dos dados, denominados de recuperadores, um registo que indica a versão dos dados copiados e um conjunto de outros registos auxiliares para implementação do protocolo de comunicação. É responsabilidade do *Front End* manter as cópias atualizadas, bem como selecionar os nós que devem possuir uma cópia.

Manutenção das cópias

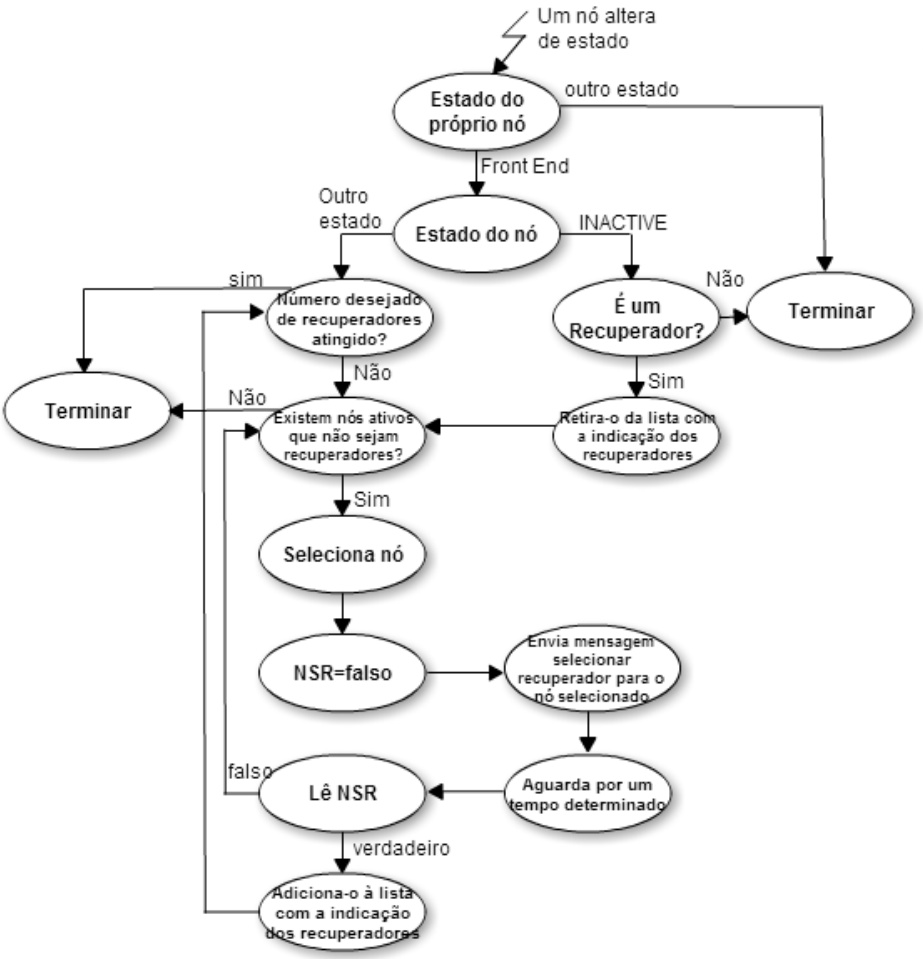

Figura 3.26 Ciclo de vida de um `Notificador` no módulo **Recuperação** na manutenção das cópias

Este módulo é um subscritor do componente **Dados Gerais** do **ITSO**, como tal é notificado sempre que um nó altera de estado. Quando tal acontece, e o próprio nó desempenha a função de *Front End,* é verificado se o nó cujo estado foi alterado é INACTIVE. Caso tal se confirme, significa que um nó deixou de pertencer à MVP, e se porventura for um recuperador, é necessário retirá-lo da lista.

Uma vez que neste momento há menos um nó com uma cópia dos dados, é necessário selecionar outro para o seu lugar. Para isso, verifica-se se há nós na MVP que ainda não possuem uma cópia. Caso tal se confirme, o *Front End* seleciona um desses nós, escreve no registo auxiliar NO_SELECIONADO_RESPONDEU (NSR) o valor `falso`, e envia a mensagem SELECIONAR_RECUPERADOR que possui uma cópia dos dados, para o nó selecionado. De

seguida aguarda por um período de tempo determinado e lê o registo NSR. Se o valor permanecer `falso` (o nó selecionado não respondeu), é verificado novamente se há mais nós ativos que ainda não sejam recuperadores. Caso exista pelo menos um nó nessas condições, repete-se o procedimento anterior desde a seleção de um nó. Caso não existam nós nessas condições o procedimento é terminado. Se o valor do registo NSR for `verdadeiro` (o nó selecionado respondeu), o nó é adicionado à lista de nós que possuem uma cópia.

Caso o nó cujo estado foi alterado não seja INACTIVE, significa que é possível que um novo nó se tenha juntado à MVP. Neste caso, é verificado se o número de nós que possuem uma cópia dos dados atingiu o valor desejado. Caso sejam necessários mais nós repete-se o processo explicado anteriormente.

Em concorrência com este processo, os recuperadores são notificados pelo *Front End* quando ocorrem determinados acontecimentos de forma a atualizarem os seus dados. Esses acontecimentos correspondem às operações que um subscritor do componente **Dados do *Front End*** deve implementar.

Acontecimentos	Operações despoletadas
quando uma nova tarefa é adicionada à fila de tarefas para execução	newTaskToExecute(task)
quando uma tarefa é retirada da fila de tarefas para execução e adicionada na fila de tarefas em execução	taskInExecution(task, dispatcher)
quando uma tarefa é retirada da fila de tarefas em execução e adicionada à fila de tarefas executadas	taskExecuted(task, dispatcher)
quando uma tarefa é retirada da fila de tarefas em execução e adicionada à fila de tarefas para execução	abortTaskExecution(dispatcher)
quando uma tarefa é retirada da fila de tarefas executadas	taskDeleted(taskID)

Tabela 3.5 Acontecimentos que provocam uma atualização das cópias

Quando uma destas operações é despoletada, o *Front End* incrementa o valor do registo `versão` e envia uma mensagem para cada um dos recuperadores com os dados associados à operação despoletada e com o valor do registo `versão`. Os recuperadores ao receberem estas mensagens atualizam o valor do registo `versão` e os dados do componente **Dados do *Front End***.

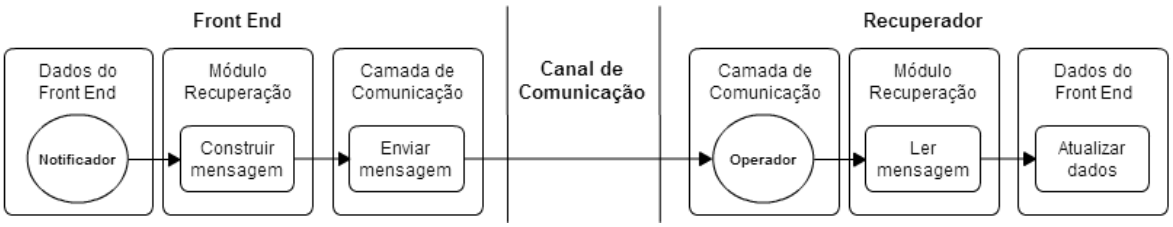

Figura 3.27 Atualização do componente **Dados do *Front End***

Recuperação de falhas

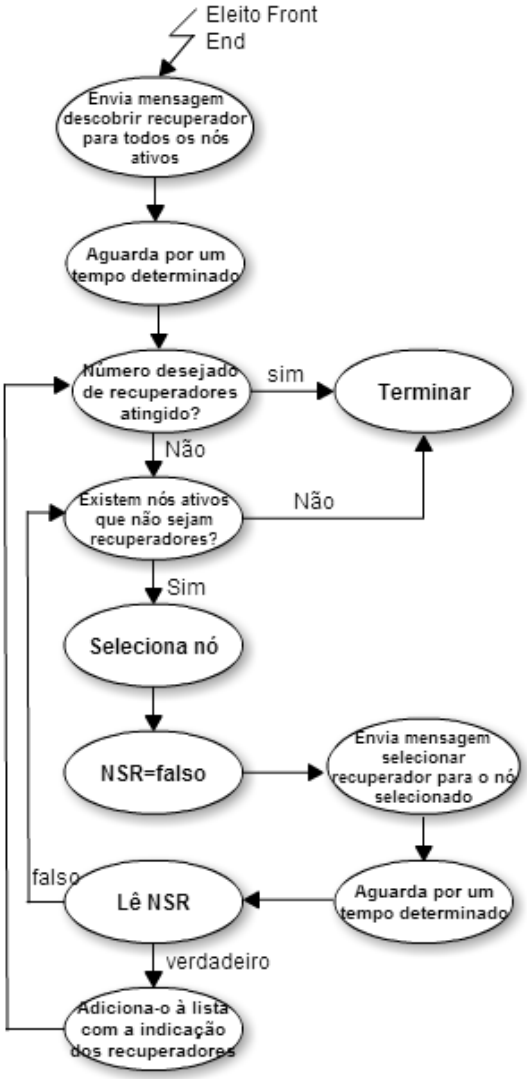

Figura 3.28 Ciclo de vida de um `Notificador` no módulo **Recuperação** na recuperação de falhas

Quando o próprio nó é eleito *Front End*, este módulo é notificado. Quando tal acontece envia a mensagem DESCOBRIR_RECUPERADOR para todos os nós ativos da MVP a pedir que os nós com uma cópia se identifiquem, e aguarda um determinado período de tempo. Após esse período, o nó assume que o componente **Dados do *Front End*** e o registo `versão` foram atualizados e a lista que identifica os recuperadores foi preenchida. Momento em que verifica se é necessário selecionar novos nós para pertencer à lista.

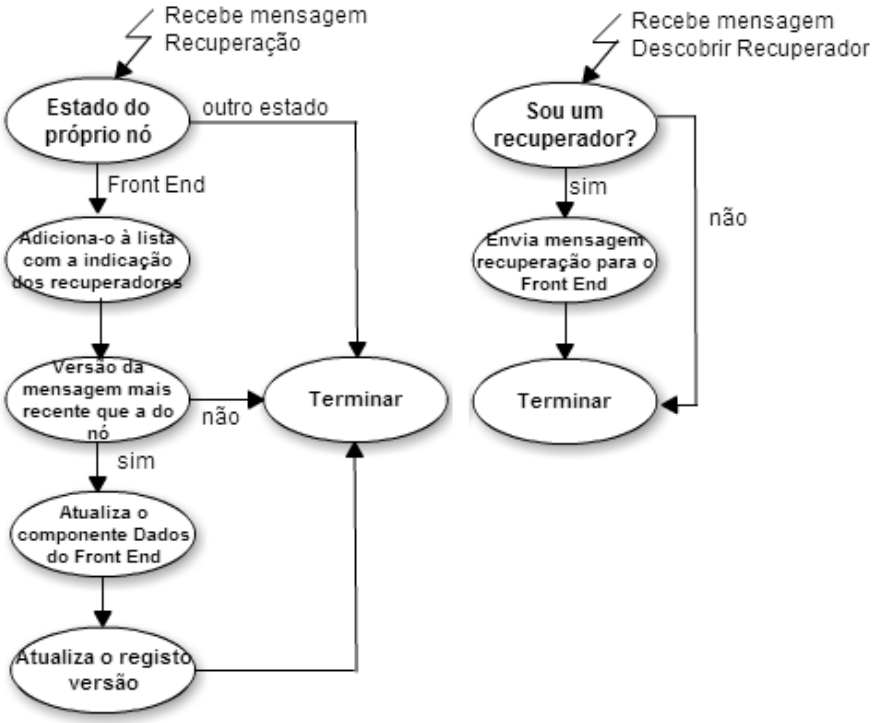

Figura 3.29 Ciclo de vida dos operadores ao receber as mensagens DESCOBRIR_RECUPERADOR e Recuperação

Quando um nó recebe a mensagem Descobrir Recuperador, e ele próprio é um recuperador, envia a mensagem Recuperação para o *Front End* com uma cópia dos **Dados do *Front End*** e a versão. Quando o *Front End* recebe esta mensagem, adiciona o nó à lista de nós que identifica os nós recuperadores, e caso a versão dos dados seja mais recente que a do nó, atualiza os dados do componente **Dados do *Front End*** e o registo `versão`.

3.2.4. Camada de Operação

Tal como na **Camada Topológica**, esta camada possui um conjunto de módulos independentes. No entanto, neste caso cada um dos módulos não implementa um determinado protocolo, mas uma fase do ciclo de vida de um nó. Essas fases são despoletadas através da API que esta camada disponibiliza. A API é constituída por quatro operações principais: `registar`,

`obter função`, `executar função` e `desregistar`. O objetivo da primeira função é registar o nó no agregado de nós da MVP, ou seja, conhecer os nós que constituem o agregado e vice-versa. A segunda operação é aguardar que o nó obtenha uma função a desempenhar no agregado (*Front End*, *Dispatcher* ou *Executor*). A terceira operação é executar uma determinada função no agregado consoante a função obtida através da operação anterior. Por último, a quarta operação tem como objetivo indicar os nós do agregado que deixará de fazer parte e de estar disponíveis para executar qualquer função.

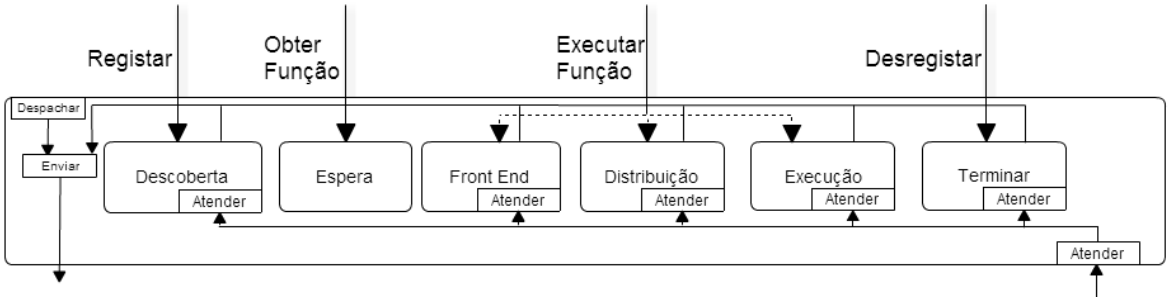

Figura 3.30. **Camada de Operação**

Cada uma destas operações atua sobre um módulo diferente, mais concretamente, a operação `registar` atua sobre o módulo **Descoberta**, a operação `obter função` sobre o módulo **Espera**, a operação *`desregistar`* sobre o módulo **Terminar**, e a operação executar função sobre os módulos ***Front End***, **Distribuição** e **Execução**, consoante a função a desempenhar.

3.2.4.1. Módulo Descoberta

Este módulo é responsável por iniciar a lista de nós do componente **Dados Gerais** do **ITSO**, e desta forma identificar que nó desempenha a função de *Front End*.

Figura 3.31. Módulo **Descoberta**

Quando é selecionada a função `registar` do nó, em primeiro lugar é alterado o estado do nó de INACTIVE para WAITING no componente **ITSO**. De seguida, é atribuído ao registo `resposta` (um registo interno do módulo) o valor `falso` e enviada para todos os nós que possivelmente pertencem ao agregado a mensagem DESCOBERTA. Após as mensagens terem sido enviadas é aguardado um determinado período de tempo. Após esse período é lido o registo `resposta`, e caso o valor continue a ser `falso` é então definido no componente **ITSO** que o *Front End* não existe. Esta definição é necessária para despoletar eventos noutros módulos, como o caso do módulo **Eleição**. Se o registo `resposta` possuir o valor `verdadeiro`, a operação é simplesmente terminada.

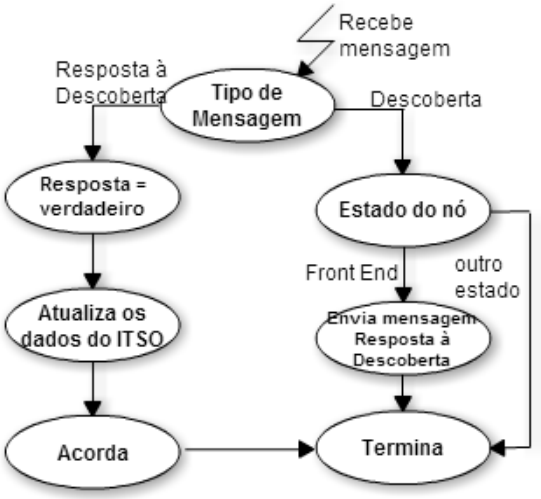

Figura 3.32 Ciclo de vida de um `Operador` no módulo **Descoberta**

Este módulo lida com dois tipos de mensagem: DESCOBERTA e RESPOSTA_DESCOBERTA. A primeira é enviada pelos nós que se querem registar na MVP, e visa identificar o nó que desempenha a função de *Front End*. Um nó ao receber esta mensagem verifica se possui o estado de *Front End,* e caso tal se verifique envia para o nó a mensagem RESPOSTA_DESCOBERTA que contém uma cópia da lista de nós do componente **Dados Gerais** do **ITSO**. Ao receber esta mensagem o nó atualiza a informação do **ITSO**, sinaliza no registo `resposta` que recebeu uma resposta atribuindo o valor `verdadeiro`, e acorda qualquer entidade que esteja à espera de uma resposta.

No fim deste processo, o nó fica a conhecer o estado dos nós que pertencem à MVP, e através do módulo **Atualização** da **Camada Topológica** é enviada uma mensagem UPDATE para o *Front End*, altura em que este toma conhecimento da disponibilidade do nó e informa os restantes desse facto.

3.2.4.2. Módulo Espera

O objetivo deste módulo é bloquear a entidade ativa que controla o nó enquanto nenhuma função é atribuída ao mesmo. Após a execução da operação `registar`, a operação `Obter Função` deve ser executada, que simplesmente bloqueia enquanto o estado do nó for WAITING. Este módulo é um subscritor do componente **`Dados Gerais`** do **ITSO**, como tal é notificado quando um nó altera de estado. Quando tal acontece relativamente ao próprio nó, se o estado deste for diferente de WAITING, então a entidade bloqueada é acordada.

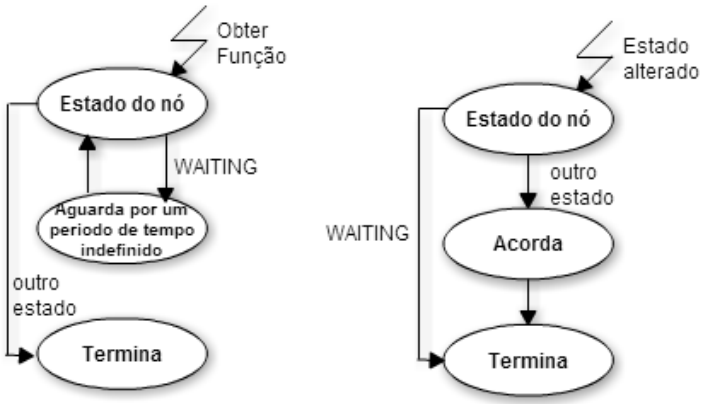

Figura 3.33. Módulo **Espera**

3.2.4.3. Módulo *Front End*

Este módulo é responsável por executar o ciclo de vida de um *Front End*, sendo as suas principais tarefas a comunicação com os clientes, e designar nós para a função de *Dispatcher*. Este módulo é ativado quando o nó possui o estado *FRONT END*, e a operação `Executar Função` da **Camada de Operação** é executada.

Figura 3.34 Ciclo de vida do módulo ***Front End***

Primeiro, é necessário selecionar uma tarefa que esteja em fila de espera para ser executada. Para isso, é invocada a operação `waitTask` do componente **ITSO**, que como já foi referido enquanto não existir uma tarefa disponível, a entidade que a chama é bloqueada.

Quando existe uma tarefa para executar, é necessário selecionar um nó para ser o seu *Dispatcher*. Primeiro é verificado se existem nós disponíveis, isto é, nós no estado WAITING. Caso não existam, a entidade é novamente bloqueada até pelo menos um nó tomar esse estado. Dos nós disponíveis é selecionado um, e é escrito no registo NO_SELECIONADO_RESPONDEU (NSR) o valor `falso`. De seguida, é enviada uma mensagem para esse nó a notificá-lo que foi selecionado para desempenhar a função de *Dispatcher*. Após enviar a mensagem, a entidade aguarda durante um determinado período de tempo antes de ler o valor do registo NSR. Caso o valor permaneça `falso` é necessário repetir o processo desde a verificação da existência de nós disponíveis, até que o valor do registo seja `verdadeiro`. Quando tal acontece, o nó selecionado é associado à tarefa no componente **ITSO**, e é-lhe enviada a mensagem TAREFA, com a tarefa correspondente. Após este procedimento a entidade volta a aguardar por uma tarefa para executar.

Durante este procedimento, a entidade ativa que o processa é bloqueada várias vezes a aguardar por um determinado acontecimento. Esses acontecimentos são despoletados pela receção de determinadas mensagens, ou determinadas alterações no componente **ITSO**. De seguida, são enumerados os vários eventos, bem como o estado que o nó deve possuir para atender a esses eventos, e a ação correspondente.

Quando o *Front End* recebe a mensagem TAREFA por parte de um cliente, adiciona a tarefa contida na mensagem à fila de espera do componente **ITSO**. Neste momento, se a entidade ativa estiver bloqueada a aguardar uma tarefa para executar é acordada.

Quando um nó no estado WAITING recebe a mensagem SELECIONADO, que indica que foi selecionado pelo *Front End* para desempenhar a função de *Dispatcher*, responde com a mensagem RESPOSTA_SELECIONADO e altera o seu estado para DISPATCHER no **ITSO**. Por sua vez, quando o *Front End* recebe a mensagem RESPOSTA_SELECIONADO, escreve no registo NSR o valor `verdadeiro` e acorda a entidade ativa se esta se encontra a aguardar uma resposta. Quando um *Dispatcher* recebe a mensagem TAREFA por parte do *Front End*, armazena a tarefa contida na mensagem no componente **Dados do *Dispatcher*** do **ITSO**.

Para além destes acontecimentos, existe outro que apesar de não influenciar a entidade ativa do módulo diretamente, é bastante importante. Quando um nó altera para um estado diferente de DISPATCHER é verificado se esse nó se encontra associado a alguma tarefa. Caso tal se verifique, significa que o estado anterior seria DISPATCHER, e que por algum motivo não terminou de distribuir a tarefa. Portanto, a tarefa é colocada novamente na lista de espera para execução.

3.2.4.4. Módulo Distribuição

Este módulo tem como objetivo implementar o ciclo de vida do *Dispatcher*, isto é, obter as subtarefas de uma tarefa e distribuí-las por vários *Executors*. Este módulo é executado quando a operação `Executar Função` da camada **operação** é chamada, e o estado do nó é DISPATCHER.

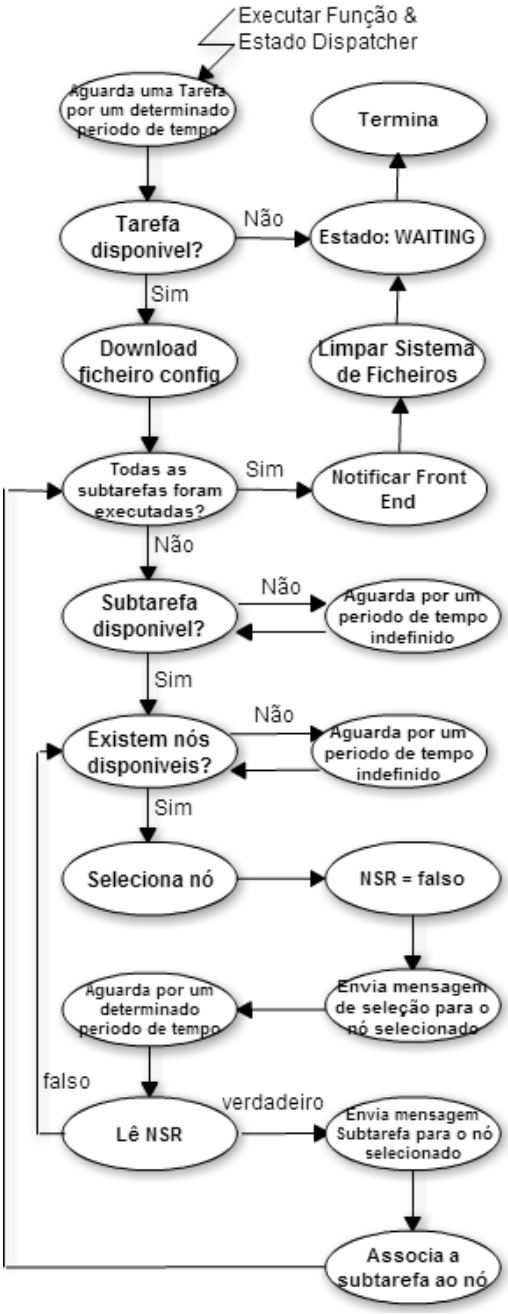

Figura 3.35 Ciclo de vida do módulo **Distribuição**

Em primeiro lugar, a entidade ativa que processa este módulo é bloqueada por um determinado período de tempo, a aguardar que uma tarefa esteja disponível no componente **Dados do *Dispatcher***. A tarefa é lá colocada por um `Operador` através do módulo ***Front End***, como referido na secção anterior.

Se ao fim desse tempo não existir uma tarefa, o nó assume que a tarefa foi perdida ou atribuída a outro nó, alterando assim o seu estado para WAITING terminando de seguida a operação. Caso contrário, é feito o *download* do ficheiro que descreve a tarefa (ficheiro *config*). Através do ficheiro são obtidas as subtarefas e como estas se encontram organizadas entre si. Quando todas as subtarefas tiverem sido executadas, é enviada uma mensagem ao *Front End* a notificá-lo que a tarefa foi executada, é removido o ficheiro config e o estado é colocado a WAITING no **ITSO**.

Enquanto existirem subtarefas por executar é verificado se existem subtarefas disponíveis. Uma tarefa encontra-se disponível quando todas as subtarefas que a precedem foram executadas. Enquanto não existirem subtarefas disponíveis a entidade é bloqueada. Das subtarefas disponíveis é selecionada uma, sendo de seguida selecionado um nó para a executar. Esta seleção é idêntica ao método de seleção de um *Dispatcher* no módulo ***Front End***. Primeiro é verificado se existem nós disponíveis (nós no estado WAITING), e enquanto tal não acontecer a entidade é bloqueada. Após selecionar um nó, é escrito o valor `falso` no registo NO_SELECIONADO_RESPONDEU (NSR), e é enviada a mensagem SELECIONADO. Após aguardar por um determinado período, a entidade lê o registo NSR, e caso o valor lido seja `falso` repete todo o procedimento desde a verificação da disponibilidade dos nós. Caso o valor seja `verdadeiro` é enviada a mensagem SUBTAREFA com a subtarefa, e o nó é associado a esta.

A entidade é bloqueada algumas vezes durante este procedimento a aguardar determinados acontecimentos. Quando um nó no estado WAITING recebe a mensagem SELECIONADO, que indica que foi selecionado para desempenhar a função de *Executor*, responde com a mensagem RESPOSTA_SELECIONADO, e altera o seu estado para EXECUTOR. Por sua vez, quando o *Dispatcher* recebe a mensagem RESPOSTA_SELECIONADO escreve no registo NSR o valor `verdadeiro`, e acorda a entidade ativa se esta se encontra a aguardar por uma resposta. Quando um *Executor* recebe a mensagem SUBTAREFA, armazena a subtarefa contida na mensagem no componente **Dados do *Executor*** do **ITSO**.

Quando um nó altera para um estado diferente de EXECUTOR é verificado se esse nó se encontra associado a alguma subtarefa. Caso tal se verifique significa que o estado anterior seria EXECUTOR, e que por algum motivo não terminou de executar a subtarefa. Por esse motivo, a subtarefa é colocada novamente na lista de espera para execução.

Sempre que a entidade que executa este procedimento é acordada verifica se o nó permanece no estado DISPATCHER. Caso tal não se confirme (por exemplo o nó foi eleito *Front End)* o procedimento é terminado, e o ficheiro `config` da tarefa eliminado do sistema de ficheiros.

3.2.4.5. Módulo Execução

O objetivo deste módulo é executar subtarefas. Este módulo é executado quando a operação `Executar Função` da camada **operação** é chamada, e o estado do nó é EXECUTOR.

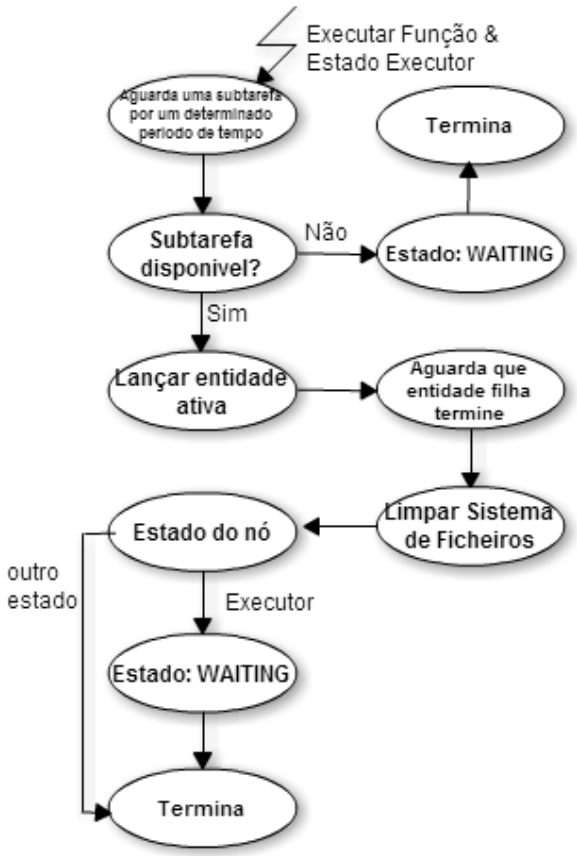

Figura 3.36 Ciclo de vida do módulo **Execução**

Em primeiro lugar, a entidade ativa que processa este módulo é bloqueada por um determinado período de tempo, a aguardar que uma subtarefa esteja disponível no componente **Dados do *Executor***. A subtarefa é colocada por um `Operador` através do módulo **Distribuição**, como referido na secção anterior.

Se ao fim desse tempo não existir uma subtarefa, o nó assume que a subtarefa foi perdida ou atribuída a outro nó, como tal altera o seu estado para WAITING, terminando de seguida a operação. Caso contrário, é lançada uma entidade ativa, e a entidade principal bloqueia até a entidade filha terminar. Após acordar, os ficheiros associados à subtarefa são eliminados do

sistema de ficheiros local, e caso o estado do nó permaneça EXECUTOR este é colocado a WAITING.

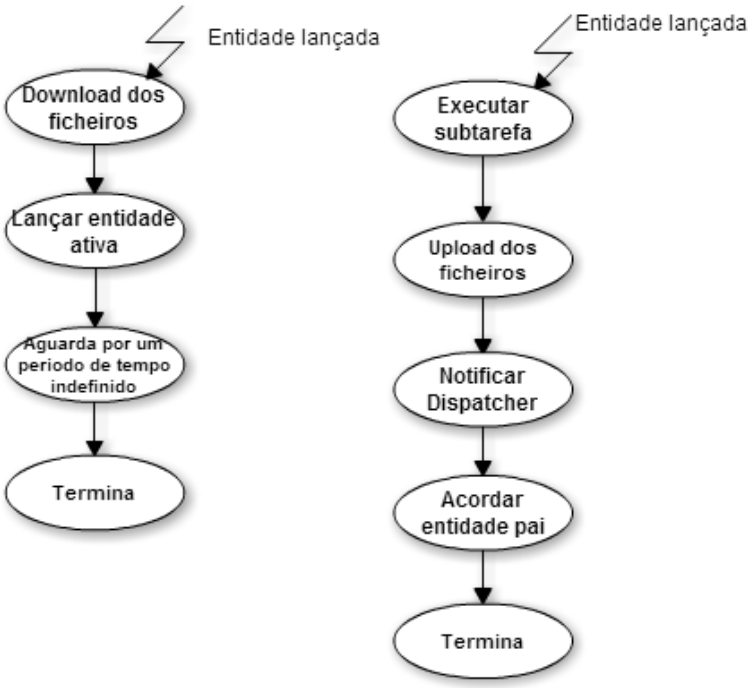

Figura 3.37 Ciclo de vida das entidades lançadas no módulo

A entidade filha, começa por fazer *download* dos ficheiros necessários para a execução da subtarefa (ficheiros jar, com dados de entrada e saída). De seguida lança uma entidade filha (neta da principal), bloqueando por um tempo indeterminado. Após acordar, termina.

A entidade neta executa a subtarefa, faz o *upload* dos ficheiros com os resultados, envia uma mensagem ao *Dispatcher* a notificá-lo que a subtarefa foi executada, e acorda a entidade que a lançou antes de terminar.

Na linguagem Java, uma entidade ativa pode ser implementada pelo tipo de dados `Thread`. Este tipo de dados não disponibiliza nenhuma operação que permita cancelar a sua execução. No entanto, na linguagem Java quando um `Thread` termina a sua execução, as entidades ativas lançadas pelo mesmo (entidades filhas) são destruídas. Tendo este aspeto em consideração, através do lançamento de duas novas entidades ativas torna-se possível abortar a execução de uma subtarefa, o que é necessário caso um *Executor* seja eleito *Front End*.

Analisando em primeiro lugar o caso de a execução de uma subtarefa ocorrer normalmente. Quando a entidade neta se encontra a executar a subtarefa, as entidades principal e filha encontram-se bloqueadas, a primeira enquanto a segunda não terminar, e a segunda indefinidamente. Quando a entidade neta termina a execução da subtarefa, acorda a entidade filha e termina. A entidade filha ao acordar simplesmente termina, acordando assim a entidade

principal. Esta limpa o sistema de ficheiros, e ao verificar que o estado do nó se mantém, altera-o para WAITING, terminando assim o ciclo de vida de um *Executor.*

Caso o *Executor* seja eleito *Front End*, este módulo é notificado devido ao facto de ser um subscritor do componente **Dados Gerais** do **ITSO**. Quando tal acontece, o `Notificador` responsável por este módulo acorda a entidade filha que termina de seguida. Ao terminar, a entidade neta é destruída, cancelando assim a execução da subtarefa, e a entidade principal é acordada. Esta por sua vez limpa o sistema de ficheiros, mas já não altera o estado para WAITING.

3.2.4.6. Módulo Terminar

O objetivo deste módulo é indicar ao *Front End* que o nó já não se encontra disponível para desempenhar qualquer função, e como tal deixará de fazer parte do agregado. Por outro lado, se o nó desempenhar a função de *Front End* notifica os restantes nós que é necessário eleger um novo *Front End*. Quando a operação `desregistar` é executada, é definido para que nós será enviada a mensagem TERMINAR consoante o estado que o nó possua. Se for *Front End,* então os destinatários da mensagem TERMINAR serão todos os nós ativos, caso contrário apenas o *Front End* é o destinatário. De seguida, é alterado o estado do nó para INACTIVE e enviada a mensagem TERMINAR para os respetivos nós.

Quando um nó recebe a mensagem TERMINAR, independentemente da função que desempenhe, altera no componente **ITSO** o estado do nó que a enviou para INACTIVE.

3.3. Serviços web

Para que os clientes do sistema não necessitem de conhecer os pormenores do protocolo de comunicação com a MVP, foi desenvolvido um conjunto de serviços *web*. Para além de encapsularem as mensagens trocadas com o *Front End*, estes serviços são responsáveis por colocar os ficheiros associados às tarefas dos clientes no **Sistema de Ficheiros Distribuído**, e por gerir uma base de dados onde são armazenadas as informações das tarefas.

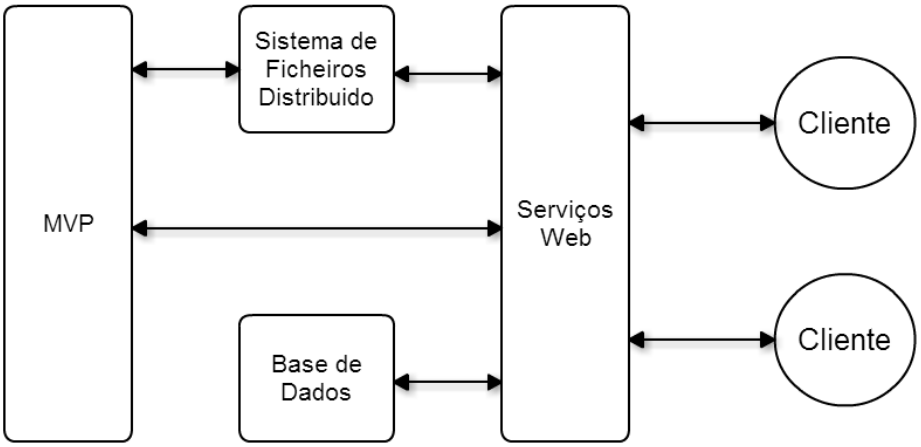

Figura 3.38 Arquitetura do sistema

3.3.1. API de acesso

As operações disponibilizadas pelos serviços *web* encontram-se divididas em três áreas: gestão de utilizadores do sistema (operações de registo, autenticação, entre outros), gestão dos nós da MVP (iniciar e parar a sua execução), e preparação das tarefas (criação de tarefas, *download* e *upload* de ficheiros, etc.).

Função	Descrição
create(username, password)	Criação de um novo utilizador do sistema com as respetivas credenciais na base de dados
login(username, password)	Autenticação de um utilizador nos sistema
update(username, password, id, role)	Alteração dos privilégios de um utilizador com um determinado ID na base de dados.
remove(username, password, id)	Invalidação de um utilizador com um determinado ID na base de dados.
users(username, password)	Obtenção da lista de utilizadores válidos na base de dados
roles()	Obtenção da lista de papéis que os utilizadores podem desempenhar na base de dados

Tabela 3.6 Serviços de gestão dos utilizadores

Função	Descrição
states(username, password)	Obtenção do estado dos nós que possivelmente pertencem à MVP
start(username, password, IP)	Iniciar a execução de um nó com um determinado endereço IP da MVP
stop(username, password, IP)	Parar a execução de um nó com um determinado endereço IP da MVP

Tabela 3.7 Serviços de gestão dos nós

Função	Descrição
create(IDuser, taskname)	Criação de uma tarefa com um determinado nome e associada a um determinado utilizador na base de dados. É criado também um diretório no **Sistema de Ficheiros Distribuído**.
state(IDTask)	Obtém o estado de uma tarefa (em espera, em execução ou executada)
upload(IDTask, filename, datastream)	Faz o *upload* de um determinado ficheiro para o diretório correspondente no sistema de ficheiros, e regista na base de dados.
download(IDTask, filename)	Faz o *download* de um determinado ficheiro do diretório correspondente no sistema de ficheiros, e regista na base de dados.
start (IDTask)	Envia a tarefa para o *Front End* para execução
destroy(IDTask)	Indica ao *Front End* que os resultados de uma tarefa já foram obtidos, e a tarefa é retirada da lista de tarefas executadas. Também é eliminado o diretório associado no sistema de ficheiros e removida a tarefa da base de dados
list(IDuser)	Lista de tarefas de um determinado utilizador
listfiles(IDTask)	Lista de ficheiros de uma determinada tarefa
Removefile(IDFile)	Remover um determinado ficheiro do sistema de ficheiros e da base de dados

Tabela 3.8 Serviços de gestão das tarefas

3.3.2. Sistema de Ficheiros Distribuído

O objetivo deste componente é encapsular o mecanismo utilizado na transferência de ficheiros, neste caso o mecanismo implementado utiliza *Remote Method Invocation* (RMI). RMI

permite invocar métodos de objetos remotos, isto é, objetos que não se encontram na máquina virtual de Java da entidade que invoca os seus métodos. De uma forma simples, RMI é constituído por um servidor e um cliente. O servidor instancia o objeto a que se pretende aceder e associa-o a um determinado porto de escuta. As mensagens recebidas por esse porto de escuta indicam que método deve ser invocado sobre o objeto e os parâmetros correspondentes. Após a invocação do método, os resultados são enviados para os clientes. No lado do cliente, os métodos são invocados sobre um *stub*, um componente que possui os mesmos métodos que o objeto remoto (a mesma interface), que envia mensagens para o servidor com a informação necessária para invocar o método. Também recebe as mensagens com os resultados que são retornados à entidade que invocou o método. As mensagens são encapsuladas pelo RMI.

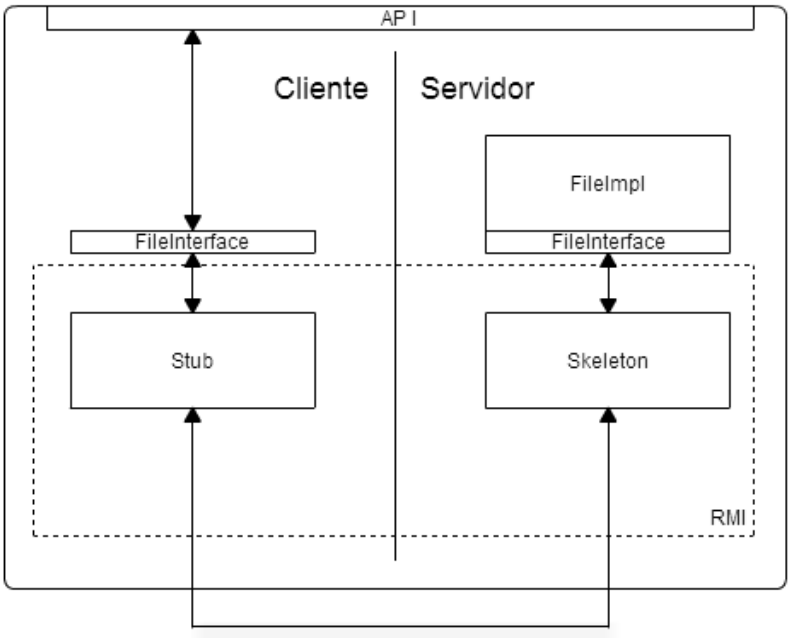

Figura 3.39 Arquitetura do **Sistema de Ficheiros Distribuído**

A figura anterior esquematiza a arquitetura deste componente, onde é possível verificar que também este componente implementa um servidor e um cliente. Como tal a API disponibiliza métodos que operam sobre o lado do servidor, e métodos que operam sobre o lado do cliente.

Função	Descrição
getInstance()	instancia um objeto da classe FileImpl e associa-o a um porto de escuta
destroyInstance()	desassocia o objeto da classe FileImpl instanciado pela operação getInstance() do porto de escuta e

	liberta a memória ocupada por o mesmo
cleanLocalFiles(files[])	elimina um determinado conjunto de ficheiros do sistema de ficheiros local
downloadFiles(remoteHost, remotePort, path[])	faz o *download* de um conjunto de ficheiros de um determinado repositório de ficheiros
uploadFiles(remoteHost, remotePort, path[])	faz o *upload* de um conjunto de ficheiros para um determinado repositório de ficheiros
cleanRemoteFiles(remoteHost, remotePort, path[])	elimina um determinado conjunto de ficheiros de um repositório de ficheiros
createRemoteDir(remoteHost, remotePort, path)	cria um diretório num repositório de ficheiros
removeRemoteDir(remoteHost, remotePort, path)	elimina um diretório de um repositório de ficheiros

Tabela 3.9 API do **Sistema de Ficheiros Distribuído**

3.3.3. Base de Dados

A base de dados do sistema armazena a informação dos utilizadores, bem como tarefas associadas a estes, e ficheiros associados às tarefas. O sistema de base de dados utilizado neste projeto foi o `sqlite`, devido à fácil portabilidade dos dados, e ao facto de ser mais "leve" que outros sistemas (por exemplo, MySQL ou SQLServer).

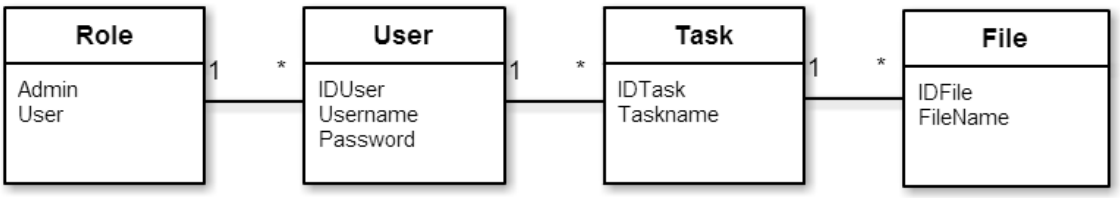

Figura 3.40 Arquitetura da base de dados

Como é possível verificar, um utilizador é apenas constituído por um *username*, *password*, e possui privilégios de administrador ou de um utilizador normal. Cada utilizador possui associado um conjunto de tarefas que são identificadas por um nome. As tarefas, por sua vez, possuem associado a si um conjunto de ficheiros, também estes identificados por um nome.

3.4. Cliente - *Website*

Foi implementado um cliente com o propósito de demonstrar o correto funcionamento de todo o sistema, e exemplificar como utilizar os serviços disponíveis. Neste caso, o cliente implementado é um *website* simples com quatro páginas: página de *login* e registo, página de tarefas, página de gestão de nós e página de gestão de utilizadores. As páginas de gestão só se encontram visíveis para utilizadores com privilégios de administrador.

Figura 3.41 Página de *login* e registo

A página inicial do *website* é a página de registo e *login*, e permite aos utilizadores registarem-se e autenticarem-se no sistema. Os utilizadores que se registarem não possuem privilégios de administrador.

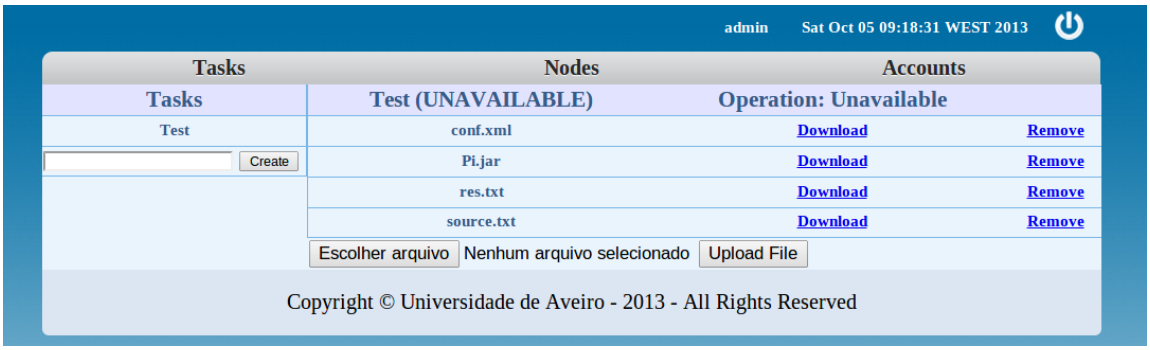

Figura 3.42 Página de tarefas

Após o *login* no sistema o cliente é redirecionado para a página de tarefas, que também pode ser acedida através da opção `Tasks` no menu do *website*. Esta página possui no lado esquerdo a lista de tarefas que o utilizador criou, sendo que na figura apenas existe a tarefa `Test`. No fim da lista existe uma caixa de texto e o botão *Create* que permitem adicionar novas tarefas à

lista. Ao selecionar uma tarefa, no lado direito são disponibilizados os detalhes da mesma. Na primeira linha e possível ver o nome da tarefa, seguido do estado da mesma:

- *UNAVAILABLE* caso não exista *Front End* na MVP.
- *NOT EXECUTED* se a tarefa ainda não foi enviada para a MVP.
- *WAITING* se a tarefa se encontra na fila de espera para execução.
- *EXECUTING* se a tarefa se encontra designada a um *Dispatcher*.
- *EXECUTED* se a tarefa já foi processada.

Consoante o estado da tarefa existe uma operação que o utilizador pode iniciar. Se o estado for *NOT EXECUTED* a operação disponível é *Execute*, caso seja *EXECUTED* a operação é *Destroy*. A operação *Execute* envia a tarefa para a MVP, enquanto a operação *Destroy* limpa os dados associados à tarefa do sistema, incluindo da memória do *Front End*, do **Sistema de Ficheiros Distribuído** e da base de dados.

As restantes linhas do lado direito constituem uma tabela de três colunas com o nome dos ficheiros que pertencem à tarefa e duas operações associadas: *download* e *remove.* A primeira operação faz o *download* do ficheiro associado, enquanto a segunda remove-o. Por baixo da tabela, existem dois botões: escolher arquivo e *upload file*. O primeiro abre uma janela de exploração, que permite ao utilizador selecionar um ficheiro. O segundo faz o *upload* do ficheiro selecionado para o servidor, adicionando-o à lista de ficheiros que pertencem à tarefa. Esta página utiliza todos os serviços disponibilizados referentes à gestão de tarefas.

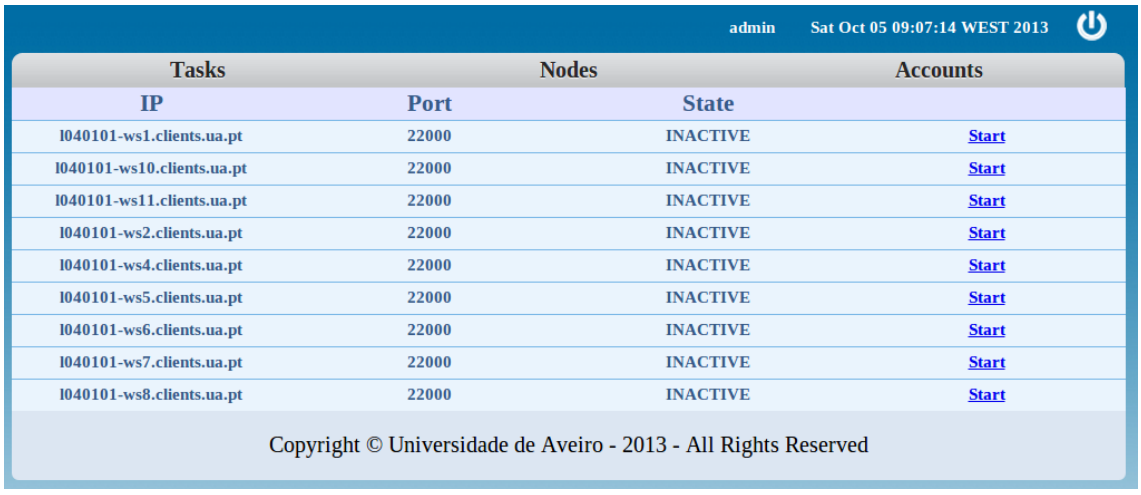

Figura 3.43 Página de gestão dos nós

Selecionando a opção *Nodes* no menu, são disponibilizados numa tabela os nós que possivelmente pertencem à MVP. A tabela é constituída por quatro colunas: endereço IP, porto de escuta, estado e operação. As duas primeiras colunas identificam os nós, a terceira coluna indica o estado do nó correspondente e a última coluna disponibiliza uma de duas operações consoante o seu estado. Se o estado do nó for *INACTIVE* então a operação disponível é *Start* que inicia o ciclo de vida do nó. Caso o nó já se encontre em funcionamento a operação disponível é *Stop* que termina o ciclo de vida do nó.

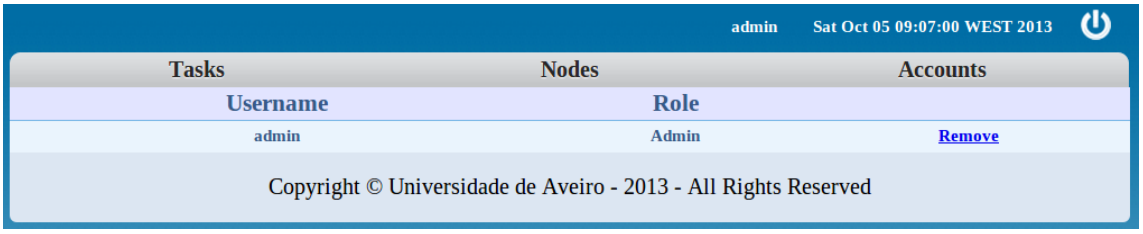

Figura 3.44 Página de gestão de utilizadores

Ao selecionar a opção *Accounts* do menu, o utilizador é redirecionado para a página de gestão de utilizadores. Esta página também é constituída por uma tabela, que possui três colunas: *username*, *role* e a operação *remove*. A coluna *username* identifica o utilizador, a coluna *role* os privilégios do utilizador (administrador ou utilizador normal), a coluna com a operação *remove* elimina o utilizador da base de dados e todas as tarefas associadas do sistema. Para alterar os privilégios de um utilizador basta clicar na célula da linha do utilizador pretendido e da coluna *role*, que provoca o aparecimento de uma *dropdown* onde é possível selecionar o papel pretendido para o utilizador.

4. Resultados

Para verificar o correto funcionamento do sistema foram executados dois tipos de testes: básico e avançado. Em ambos os testes foi utilizado um conjunto de sistemas computacionais semelhantes com o sistema operativo Linux, versão do kernel 3.8.9-200.fc18.i686.PAE. A versão da máquina virtual Java utilizada foi JDK 1.7.

4.1. Teste básico

Neste teste foram utilizados cinco sistemas computacionais, e em cada um dos sistemas foi lançado um nó, em que o tempo de vida de cada um variava aleatoriamente entre quatro e sete minutos. Ao fim desse período o processo era interrompido e lançado novamente após um período aleatório entre um e dois minutos. O teste teve uma duração aproximada de cinco horas, e o objetivo era verificar se os nós se adaptavam corretamente às alterações da MVP. Durante esse período, sempre que ocorria uma alteração do estado de um nó no componente **Dados Gerais** era escrito num ficheiro o estado de todos os nós da MVP. Desta forma, foi possível verificar, após as cinco horas, que a informação do componente **Dados Gerais** dos vários nós se manteve coerente ao longo desse período, através da comparação destes ficheiros.

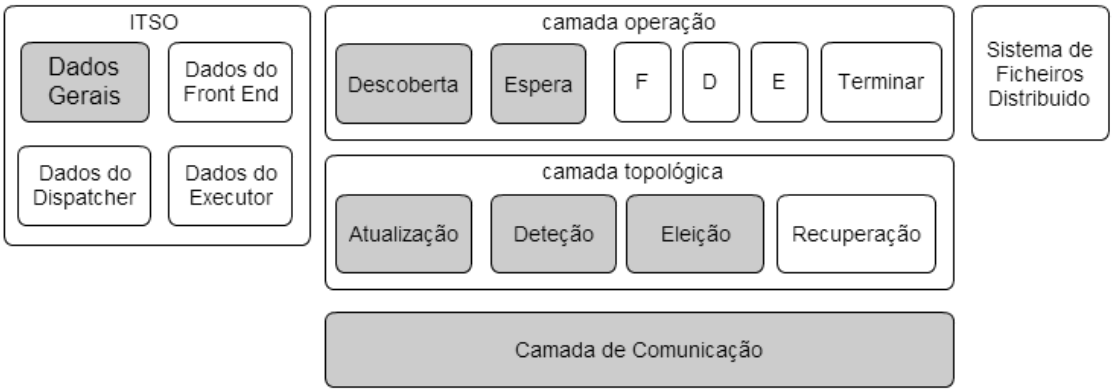

Figura 4.1 Componentes testados com fundo cinza

Este teste permitiu avaliar o funcionamento de vários componentes. Como os nós conseguiram trocar mensagens entre si, ficou demonstrado que a **Camada de Comunicação** funcionava corretamente. Sempre que um nó era lançado, detetava que nó desempenhava a função de *Front End* através do módulo **Descoberta** da **Camada de Operação** e bloqueava no módulo espera da mesma camada. Por outro lado, sempre que um nó se juntava ao agregado, os restantes eram informados através do módulo **Atualização** da **Camada Topológica**. Quando o processo que controlava um nó era interrompido, essa falha era detetada pelo módulo **Deteção** e comunicada aos restantes nós novamente pelo módulo **Atualização**. Quando o nó que falhava desempenhava a função de *Front End*, era eleito um nó para o seu lugar através do módulo

Eleição. O componente **Dados Gerais** do **ITSO** demonstrou funcionar corretamente, uma vez que os componentes anteriores "comunicavam" através dele.

Sistema Computacional	A	B	C	D	E
Prioridade (1 menos prioritário, 5 mais prioritário)	1	3	5	4	2

Tabela 4.1 Ordenação dos nós relativamente às suas prioridades

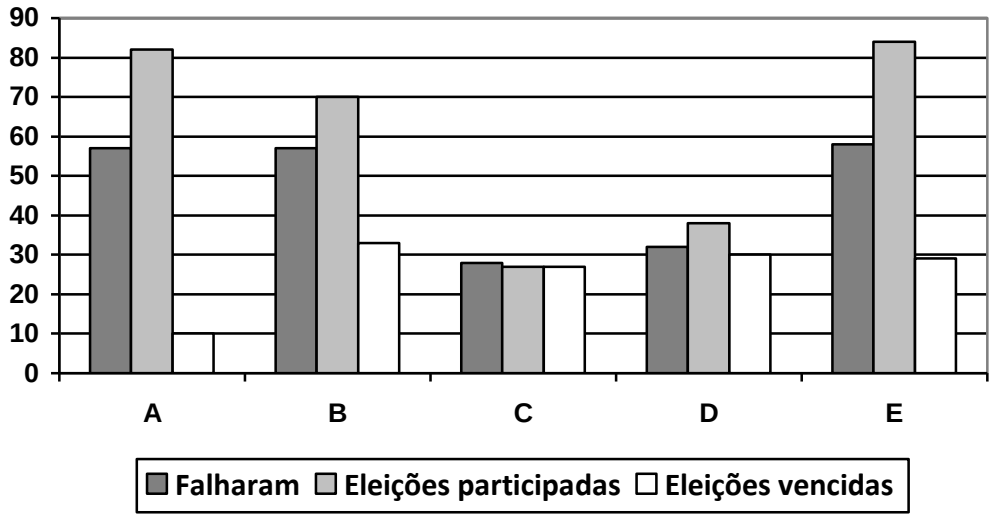

Figura 4.2 Quantidade de vezes que os nós falharam e participaram em eleições

A figura anterior ilustra a quantidade de vezes que os nós falharam, e a quantidade de vezes que participaram em eleições. Os nós A, B e E falharam perto de 60 vezes, enquanto os nós C e D falharam em média metade. É possivel verificar que o nó C ganhou todas as eleições em que participou por ser o nó com maior prioridade. O nó D ganhou quase todas as eleições, uma vez que perdia apenas as eleições em que o nó C participava. Já o nó A apenas ganhou 12% das eleições em que participou, uma vez que possui a menor prioridade de todos, ganhando apenas as eleições em que apenas ele participava.

4.2. Teste avançado

Neste teste foram usados quatro sistemas computacionais: um disponibilizava os serviços, o cliente e o **Sistema de Ficheiros Distribuído**; os restantes possuiam um nó. O objetivo deste

teste era verificar o correto funcionamento de todo o sistema: execução de uma tarefa, deteção e recuperação de falhas.

A tarefa utilizada apenas possui uma subtarefa, que lê um ficheiro de texto onde cada linha deverá possuir um número inteiro. Para cada linha, calcula o valor de Pi com o número de casas decimais igual ao número lido, e escreve os resultados noutro ficheiro. Após terminar aguarda por um "*enter*" na linha da linha de comandos, de forma a permitir testar como o sistema responde a diversos acontecimentos enquanto uma subtarefa está a ser executada. A tabela seguinte mostra os vários acontecimentos introduzidos durante o teste, o estado de cada nó após cada acontecimento e pequenas observações para ajudar a perceber o que o estado do sistema.

Eventos	Nó A	Nó B	Nó C	Observações
	INACTIVE	INACTIVE	INACTIVE	Situação inicial
Lançamento do sistema A	WAITING	INACTIVE	INACTIVE	Nó A dá início a uma eleição
	FRONT END	INACTIVE	INACTIVE	Nó A ganha a eleição
Lançamento da tarefa	*FRONT END*	INACTIVE	INACTIVE	(1 tarefa para execução no nó A)
Lançamento do sistema B	*FRONT END*	WAITING	INACTIVE	Nó B é selecionado para recuperador (1 tarefa para execução nos nós A e B)
	FRONT END	DISPATCHER	INACTIVE	Nó B selecionado para *Dispatcher* (1 tarefa em execução nos nós A e B)
Lançamento do sistema C	*FRONT END*	DISPATCHER	WAITING	
	FRONT END	DISPATCHER	EXECUTOR	Nó C selecionado para *Executor* e bloqueia
Interrupção do sistema C	*FRONT END*	DISPATCHER	INACTIVE	Falha do *Executor*
Lançamento do sistema C	*FRONT END*	DISPATCHER	WAITING	
	FRONT END	DISPATCHER	EXECUTOR	Nó C selecionado para *Executor* e bloqueia
Interrupção do sistema B	*FRONT END*	INACTIVE	EXECUTOR	Falha do *Dispatcher*
	FRONT END	INACTIVE	WAITING	Nó C selecionado para

				recuperador (1 tarefa para execução nos nós A e C)
	FRONT END	INACTIVE	DISPATCHER	Nó C é selecionado para *Dispatcher* (1 tarefa em execução nos nós A e C)
Lançamento do sistema B	*FRONT END*	WAITING	DISPATCHER	
	FRONT END	EXECUTOR	DISPATCHER	Nó B selecionado para *Executor* e bloqueia
Interrupção do sistema A	INACTIVE	EXECUTOR	DISPATCHER	Falha do *Front End*
	INACTIVE	*FRONT END*	DISPATCHER	Nó B eleito *Front End*, recupera a informação através do nó C (1 tarefa em execução nos nós B e C)
Interrupção do sistema C	INACTIVE	*FRONT END*	INACTIVE	Falha do *Dispatcher* (1 tarefa para execução no nó B)
Lançamento do sistema A	WAITING	*FRONT END*	INACTIVE	Nó A selecionado para recuperador (1 tarefa para execução nos nós A e B)
	DISPATCHER	*FRONT END*	INACTIVE	Nó A selecionado para *Dispatcher* (1 tarefa em execução nos nós A e B)
Lançamento do sistema C	DISPATCHER	*FRONT END*	WAITING	
	DISPATCHER	*FRONT END*	EXECUTOR	Nó C selecionado para *Executor* e bloqueia
Desbloquear o sistema C	DISPATCHER	*FRONT END*	WAITING	Nó C termina de executar a subtarefa
	WAITING	*FRONT END*	WAITING	Nó B distribui todas as subtarefas

Tabela 4.2 Acontecimentos introduzidos durante o teste avançado e a resposta do sistema

Inicialmente os nós ainda não tinham sido lançados, portanto possuíam o estado INACTIVE. O primeiro nó a ser lançado foi o A. Este nó altera o seu estado para WAITING, deteta que não existe um *Front End* através do módulo **Descoberta**, e dá início a uma eleição. Como é o único nó ativo não recebe mensagens CANDIDATO, alterando o seu estado para *FRONT END*.

De seguida é efetuado o pedido de executar uma tarefa ao *Front End* através do envio da mensagem TAREFA. Ao receber esta mensagem, o *Front End* insere a tarefa nela contida, na lista de mensagem para executar no componente **Dados do *Front End***. A seguir é lançado o nó do B, que altera o seu estado para WAITING e deteta que existe um *Front End* através do módulo **Descoberta**. Após descobrir que nó desempenha a função de *Front End*, envia a mensagem UPDATE para o mesmo através do módulo **Atualização**, para que o *Front End* perceba que o nó B se encontra no estado WAITING.

Após detetar que B se encontra no estado WAITING, o *Front End* seleciona-o para *Dispatcher* da tarefa recebida, através do módulo ***Front End***, e para recuperador através do módulo r**ecuperação**. Nesta fase ambos os nós possuem no componente **Dados do *Front End*** uma tarefa na fila de espera das tarefas para execução. Após B se tornar *Dispatcher*, altera o seu estado para DISPATCHER e recebe a mensagem TAREFA com a tarefa a distribuir. O *Front End* retira a tarefa da lista de tarefas para execução e adiciona-a na lista de tarefas em execução associada ao nó B. O módulo **Recuperação** notifica o nó B dessa alteração, possuindo assim ambos os sistemas no componente **Dados do *Front End*** uma tarefa na lista de tarefas em execução.

De seguida é lançado o nó C, que altera o seu estado para WAITING e deteta que nó desempenha a função de *Front End* pelo módulo **Descoberta**. Através do módulo **Atualização** notifica o *Front End* que agora se encontra no estado WAITING, que por sua vez notifica o nó B, o *Dispatcher*. O *Dispatcher* ao perceber que existe um nó no estado WAITING seleciona-o para *Executor* através do módulo **Distribuição**. Após se tornar *Executor* recebe a mensagem SUBTAREFA com a subtarefa, e executa-a, o que faz com que bloqueie. Neste momento todo o sistema se encontra estabilizado e aguardar que C termine de executar a subtarefa.

Nesta altura, o nó C é interrompido, para simular a falha de um *Executor*. O *Front End* deteta a falha deste nó através do módulo **Deteção** e comunica-a ao nó B através do módulo **Atualização**. O *Dispatcher* ao perceber que um *Executor* responsável por uma subtarefa sua falha verifica que a subtarefa não foi executada com sucesso e coloca-a novamente disponível para execução. O nó C é novamente lançado, e tal como antes, é selecionado para *Executor*. O sistema estabiliza novamente a aguardar que C termine a execução da subtarefa.

Desta vez é simulada a falha de um *Dispatcher*, interrompendo o nó B. O *Front End* deteta essa falha através do módulo **Deteção** e comunica-a ao *Executor* através do módulo **Atualização**. Neste momento, C verifica que o seu *Dispatcher* falhou, interrompendo de seguida a execução da subtarefa e altera o seu estado para WAITING. Em simultâneo, o *Front End* deteta através do

módulo **Recuperação** que o recuperador falhou, e seleciona C para tomar o seu lugar. Através do módulo ***Front End***, o *Front End* coloca a tarefa que se encontrava em execução associada ao nó B para execução. O módulo **Recuperação** notifica C que a tarefa agora se encontra na lista de espera para execução. Neste momento, os nós A e C possuem no componente **Dados do *Front End*** uma tarefa na lista de tarefas para execução. Como o *Front End* agora possui uma tarefa a aguardar para ser executada e um nó no estado WAITING, seleciona-o para *Dispatcher*. A tarefa passa para a lista de tarefas em execução, e notifica C através do módulo **Recuperação** dessa alteração. Agora ambos os nós possuem uma tarefa em execução no componente **Dados do *Front End***.

De seguida é lançado novamente o nó B, que se torna *Executor* pelos motivos explicados anteriormente, altura em que o sistema estabiliza a aguarda que o nó B termine de executar a subtarefa.

Por último é simulada a falha do *Front End*, interrompendo o nó A. Os restantes nós detetam essa falha através do módulo **Deteção**, e dão início a uma eleição através do módulo **Eleição**. Como o nó B possui o estado EXECUTOR e o nó C o estado DISPATCHER, B possui mais prioridade do que C. É por este motivo que B altera o seu estado para *FRONT END*. Ao alterar de estado, a execução da subtarefa é cancelada e começa a executar o módulo ***Front End***. O módulo **Recuperação** envia uma mensagem para os nós ativos (nó C) a perguntar quem são os recuperadores. C responde enviando os dados que possui no componente **Dados do *Front End***, uma tarefa em execução. Neste momento, B possui o mesmo estado interno que A possuía antes de falhar. Para confirmar que B possui no componente **Dados do *Front End*** uma tarefa em execução associada ao nó C, interrompeu-se C. Ao detetar essa falha, o *Front End* deve colocar a tarefa na fila de tarefas para execução e associá-la ao próximo nó que se juntar à MVP. Sendo assim, foi lançado novamente o nó A, que tal como previsto, foi selecionado para *Dispatcher*. Também foi o nó selecionado para recuperador, como tal ambos os nós possuem no componente **Dados do *Front End*** uma tarefa em execução.

De seguida foi lançado o nó C, que se tornou mais tarde *Executor*, e bloqueou a aguardar por um "*enter*" na linha de comandos como aconteceu anteriormente. Desta vez, o *Executor* foi desbloqueado, notificando de seguida o *Dispatcher* que a subtarefa tinha sido executada, e alterando o seu estado para WAITING. O *Dispatcher* notificou o *Front End* que a tarefa tinha sido executada e alterou também o seu estado para WAITING.

Através do cliente foi efetuado o *download* do ficheiro com os resultados e verificou-se que estes estavam corretos. Como tal, o sistema conseguiu recuperar das diversas falhas introduzidas para testar o seu funcionamento, e executar com sucesso a tarefa pedida.

5. Conclusões

Este projeto tinha como objetivo especificar e implementar um sistema que disponibilizasse através de um conjunto de serviços um ambiente de execução distribuído de aplicações. Este ambiente teria como recursos um agregado heterogéneo de sistemas computacionais, que poderiam ser utilizados para diversos fins durante determinados períodos de tempo. Como tal, o principal objetivo deste ambiente é a capacidade de paralelização de aplicações de forma a tornar a sua execução mais eficiente, em situações em que parte dos recursos presentes se podem tornar inesperadamente indisponíveis.

Vai fazer-se uma análise conclusiva do trabalho desenvolvido, avaliando criticamente o cumprimento dos objetivos. Para além disso, são propostas algumas funcionalidades que seriam interessantes acrescentar para melhorar o desempenho do sistema.

Como o capítulo dos **Resultados** evidencia, o sistema implementado é capaz de resistir à falha de múltiplos nós, independentemente do estado em que se encontram, mas o grau de resistência não é completo na versão atual do sistema. Numa situação limite, quando o nó que desempenha a função de *Front End* e os nós que possuem uma cópia de segurança dos dados falharem em simultâneo, a informação é perdida permanentemente. O problema pode ser minimizado fazendo com que todos os nós possuam uma cópia dos dados, mas ainda assim, a possibilidade de perda de dados continua a existir. A solução para este problema que foi pensada, consiste em contemplar um segundo nível de *backup* de forma a que na ocorrência de uma falha total, seja possível recuperar a informação quando o sistema for reiniciado.

Também no capítulo anterior foi ilustrada a execução de uma tarefa constituída por uma única subtarefa. Embora na versão atual não seja possível executar uma tarefa constituída por múltiplas subtarefas, a extensão do sistema para contemplar este tipo de situações é relativamente trivial. Só não foi feito por falta de tempo. Para o conseguir basta alterar o módulo **Distribuição** de forma a fundir os dados produzidos pela execução das várias substarefas.

Escolheu-se o protocolo TCP para a troca de mensagens entre os nós da MVP, por este protocolo garantir que as mensagens trocadas são entregues. No entanto, nada impede que seja utilizado em alternativa o protocolo UDP. Tratando-se de um protocolo mais leve, é fortemente provável que o desempenho de comunicações obtido se torne melhor. No entanto, será sempre necessária a implementação de um protocolo que garanta a entrega das mensagens trocadas.

O princípio geral de sincronização utilizado baseia-se no pressuposto de que existe um limite superior para o tempo de transmissão das mensagens trocadas. Neste sentido, os algoritmos estabelecidos assumem que os `threads` intervenientes devem bloquear por um período pré-estabelecido de tempo, enquanto aguardam a receção das respostas a iniciativas por eles despoletadas. Exemplos desses procedimentos são: o processo de eleição de um *Front End*, o processo de seleção de um nó para desempenhar a função de *Dispatcher* ou *Executor*, ou o

processo de seleção de um nó para possuir uma cópia dos dados do componente **Dados do *Front End***. Seria por isso interessante executar vários testes, o que não foi feito por falta de tempo, com uma elevada quantidade de nós, para determinar o tempo médio que um nó demora a receber a resposta a uma determinada mensagem. Desta forma, seria possível diminuir os tempos de espera, aumentando o desempenho destes procedimentos.

O sistema implementado não lida com aspetos de segurança, sendo necessário numa versão futura contemplar esta área a dois níveis: encriptação das mensagem trocadas e as permissões das aplicações executadas pelos nós. O primeiro nivel é trivial e basta cifrar e decifrar as mensagens na camada de comunicação antes de as enviar e depois de as receber, respetivamente. O segundo nível já é mais complexo lidando com as permissões que o sistema operativo oferece as aplicações, e com a deteção de códgo malicioso.

Por último, o cliente implementado visa demonstrar de uma forma mais elucidativa o correto funcionamento do sistema, e de como utilizar os vários serviços disponíveis. Para além de ser necessário melhorar o *interface* gráfico, seria também interessante incorporar uma aplicação que permitisse gerar automaticamente o ficheiro config das tarefas por interação gráfica.

6. Referências

[Akamai, 2013] *Volume 6, Number 1, The State of the Internet, 1st Quarter, 2013 Report.*

[Coulouris, Dollimore, & Kindberg, 2003] Coulouris, G., Dollimore, J., & Kindberg, T. *Distributed Systems: Concepts and Design.*

[Foster, 2002] Foster, I. (20 de Julho de 2002). *What is the Grid? A Three Point Checklist.*

[Foster, Zhao, Raicu, & Lu] Foster, I., Zhao, Y., Raicu, I., & Lu, S. (s.d.). *Cloud Computing and Grid Computing 360-Degree Compared*.

[ITU, 2013] Documentos retirados do website a 24 de Julho de 2013: http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2013/ITU_Key_2005-2013_ICT_data.xls http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2013/Individuals_Internet_2000-2012.xls

[JVM, 2013] Informação retirada do website a 24 de Julho de 2013: http://docs.oracle.com/javase/specs/jvms/se7/html/

[Li, Yang, Kandula, & Zhang, 2010] Li, A., Yang, X., Kandula, S., & Zhang, M. (2010). CloudCmp: Comparing Public Cloud Providers. 1-14.

[NIST, 2011] Mell, P., & Grance, T. (September de 2011). *The NIST Definition of Cloud Computing.*

[PVM, 1994] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., & Sunderam, V. *Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*

[Sterling, 2001] Sterling, Tomas. (2001). *Beowulf Cluster Computing with Linux.* MIT Press.

[Tanenbaum & Steen, 2006] Tanenbaum, A. S., & Steen, M. V. *Distributed Systems: Principles and Paradigms.*

[UA, 2011] Documento retirado do website a 16 de Julho de 2013: http://www.ua.pt/ReadObject.aspx?obj=19113